

Rational Application Developer



# EGL Reference Guide

*Version 6 Release 01*



Rational Application Developer



# EGL Reference Guide

*Version 6 Release 01*

**Note**

Before using this information and the product it supports, read the information in "Notices," on page 1223.

**Fourth Edition (August 2005)**

This edition applies to version 6, release 0, modification 1 of Rational Web Developer and Rational Application Developer and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 1996, 2005. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.



---

# Contents

## Overview . . . . . 1

Introduction to EGL . . . . .	1
What's new in EGL 6.0.1 . . . . .	1
What's new in EGL 6.0.0.1 . . . . .	3
What's new in the EGL 6.0 iFix . . . . .	4
What's new in EGL version 6.0 . . . . .	5
Development process . . . . .	9
Troubleshooting . . . . .	10
Out of date .jar files . . . . .	10
Improving performance at development time . . . . .	10
Runtime configurations . . . . .	11
Use of a Java wrapper . . . . .	11
Valid calls . . . . .	12
Valid transfers . . . . .	13
Sources of additional information on EGL . . . . .	14

## EGL language overview . . . . . 15

EGL projects, packages, and files . . . . .	15
EGL project . . . . .	15
Package . . . . .	16
EGL files . . . . .	16
Recommendations . . . . .	17
Parts. . . . .	19
References to parts . . . . .	23
Fixed structure . . . . .	27
Typedef. . . . .	28
Import . . . . .	33
Background . . . . .	33
Format of the import statement. . . . .	33
Primitive types . . . . .	34
Primitive types at declaration time . . . . .	36
Relative efficiency of different numeric types . . . . .	36
ANY . . . . .	37
Character types . . . . .	38
DateTime types . . . . .	41
LOB types . . . . .	48
Numeric types . . . . .	50
Declaring variables and constants in EGL . . . . .	53
Declaring a record that redefines another . . . . .	54
Dynamic and static access . . . . .	55
Scoping rules and "this" in EGL . . . . .	57
References to variables in EGL . . . . .	59
Bracket syntax for dynamic access. . . . .	61
Abbreviated syntax for referencing fixed structures . . . . .	62
Overview of EGL properties. . . . .	64
Complex properties. . . . .	66
Field-presentation properties . . . . .	67
Formatting properties . . . . .	67
SQL item properties . . . . .	68
Validation properties . . . . .	68
Set-value blocks . . . . .	68
Set-value blocks for elementary situations . . . . .	69
Set-value blocks for a field of a field . . . . .	70
Use of "this" . . . . .	71

Set-value blocks, arrays, and array elements . . . . .	72
Example with a complex property. . . . .	73
Additional examples . . . . .	73
Arrays . . . . .	75
Dynamic arrays . . . . .	75
Structure-field arrays . . . . .	79
Dictionary . . . . .	83
Dictionary properties . . . . .	84
Dictionary functions . . . . .	85
ArrayDictionary . . . . .	87
EGL statements . . . . .	88
Keyword statements in alphabetical order . . . . .	91
Transfer of control across programs . . . . .	93
Exception handling. . . . .	94
try blocks . . . . .	94
System exceptions . . . . .	95
Limits of try blocks. . . . .	95
Error-related system variables . . . . .	96
I/O statements . . . . .	97
Error identification . . . . .	98
Event handling in EGL . . . . .	99

## Migrating EGL code to EGL V6.0.1 101

## Migrating EGL code to EGL V6.0 iFix 103

EGL-to-EGL migration . . . . .	104
Changes to properties during EGL-to-EGL migration. . . . .	107
Setting EGL-to-EGL migration preferences. . . . .	112

## Setting up the environment . . . . . 115

Setting EGL preferences . . . . .	115
Setting preferences for text . . . . .	115
Setting preferences for the EGL debugger . . . . .	116
Setting the default build descriptors . . . . .	118
Setting preferences for the EGL editor . . . . .	118
Setting preferences for source styles . . . . .	119
Setting preferences for templates . . . . .	119
Setting preferences for SQL database connections . . . . .	121
Setting preferences for SQL retrieve . . . . .	123
Enabling EGL capabilities . . . . .	124

## Beginning code development . . . . . 127

Creating a project . . . . .	127
Creating an EGL project . . . . .	127
Creating an EGL Web project . . . . .	127
Specifying database options at project creation . . . . .	129
Creating an EGL source folder. . . . .	129
Creating an EGL package . . . . .	130
Creating an EGL source file . . . . .	130
Using the EGL templates with content assist . . . . .	131
Keyboard shortcuts for EGL . . . . .	131

## Developing basic EGL source code 133

Creating an EGL dataItem part . . . . .	133
DataItem part . . . . .	133
Editing a dataItem part with the source assistant	134
Creating an EGL record part . . . . .	135
Record parts. . . . .	135
Fixed record parts. . . . .	136
Non-fixed record parts . . . . .	137
Record types and properties . . . . .	138
PCB record part properties . . . . .	145
Creating an EGL program part . . . . .	147
Program part . . . . .	148
Creating an EGL function part. . . . .	149
Function part . . . . .	150
Creating an EGL Interface part . . . . .	150
EGL interfaces . . . . .	151
Creating an EGL Service part . . . . .	157
EGL services and Web services . . . . .	158
Best practices for services and related interfaces	
in EGL . . . . .	162
Web transaction support in EGL . . . . .	163
Creating an EGL library part . . . . .	169
Library part of type basicLibrary . . . . .	169
Library part of type nativeLibrary . . . . .	171
Library part of type ServiceBindingLibrary . . . . .	172
Creating an EGL dataTable part . . . . .	175
DataTable . . . . .	176

## Inserting code snippets into EGL and JSP files 179

Setting the focus to a form field . . . . .	180
Testing browsers for a session variable . . . . .	180
Retrieving the value of a clicked row in a data table . . . . .	181
Updating a row in a relational table. . . . .	182

## Working with text and print forms 183

Creating an EGL formGroup part. . . . .	183
FormGroup part . . . . .	183
Form part . . . . .	184
Creating an EGL print form . . . . .	185
Creating an EGL text form . . . . .	187
EGL form editor overview . . . . .	194
Editing form groups with the EGL form editor . . . . .	195
Creating a filter . . . . .	196
Creating a form in the EGL form editor . . . . .	196
Creating a constant field. . . . .	197
Creating a variable field in a print or text form	198
Setting preferences for the EGL form editor	
palette entries . . . . .	200
Form templates in the EGL form editor. . . . .	200
Display options for the EGL form editor . . . . .	204
Setting preferences for the EGL form editor . . . . .	204
Setting bidirectional text preferences for the	
EGL form editor . . . . .	205
Form filters in the EGL form editor . . . . .	206

## Creating a user interface with ConsoleUI. 207

Console user interface . . . . .	207
----------------------------------	-----

Creating an interface with ConsoleUI . . . . .	208
ConsoleUI parts and related variables . . . . .	209
Window . . . . .	210
Prompt . . . . .	210
ConsoleField . . . . .	210
ConsoleForm . . . . .	211
Use of new in ConsoleUI . . . . .	212
ConsoleUI screen options for UNIX . . . . .	213

## Creating an EGL Web application. 215

Web support . . . . .	215
Use of an EGL program in a Web application. . . . .	215
Programmatic control and the VGUI record . . . . .	216
Creating a single-table EGL Web application . . . . .	217
EGL Data Parts and Pages wizard . . . . .	217
Creating a single-table EGL Web application . . . . .	218
Defining Web pages in the EGL Data Parts and	
Pages wizard . . . . .	219
Creating an EGL pageHandler part . . . . .	221
Page Designer support for EGL . . . . .	221
PageHandler . . . . .	223
Supporting multiple languages for labels or help	
text in a PageHandler . . . . .	227
JavaServer Faces controls and EGL . . . . .	228
Creating an EGL field and associating it with a	
Faces JSP . . . . .	242
Associating an EGL record with a Faces JSP . . . . .	243
Binding a JavaServer Faces command	
component to an EGL PageHandler . . . . .	244
Using the Quick Edit view for PageHandler	
code . . . . .	244
Binding a JavaServer Faces input or output	
component to an EGL PageHandler . . . . .	245
Binding a JavaServer Faces check box	
component to an EGL PageHandler . . . . .	246
Binding a JavaServer Faces single-selection	
component to an EGL PageHandler . . . . .	247
Binding a JavaServer Faces multiple-selection	
component to an EGL PageHandler . . . . .	248
Segmentation in Web transactions . . . . .	250

## Creating EGL Reports. 251

EGL reports overview . . . . .	251
EGL report creation process overview . . . . .	252
JasperReports design file . . . . .	253
Report driver . . . . .	253
Optional report handler . . . . .	253
Creating the report design file. . . . .	254
EGL source files of type	
DataSource.databaseConnection . . . . .	254
EGL source files of type	
DataSource.sqlStatement. . . . .	255
EGL source files of type DataSource.reportData	255
Compiling the XML design file source . . . . .	255
Data types in XML design documents . . . . .	256
Writing code to drive a report. . . . .	257
Using report templates . . . . .	259
Sample code for EGL report-driver functions. . . . .	260
Report and ReportData parts . . . . .	262
Creating subreports . . . . .	263

EGL report handler . . . . .	264
Responding to events. . . . .	265
Creating explicitly invoked functions . . . . .	265
Storing and retrieving data . . . . .	265
Generated output . . . . .	266
Creating an EGL report handler . . . . .	266
Report handler template. . . . .	267
Getting report parameters . . . . .	268
Getting and setting report variables . . . . .	268
Getting report field values . . . . .	269
Saving report data in the report handler . . . . .	269
Retrieving report data in the XML file . . . . .	269
Invoking a function from the XML design document . . . . .	270
Predefined EGL report handler functions . . . . .	271
Additional EGL report handler functions . . . . .	272
Function for getting report parameters . . . . .	272
Functions for setting and getting report variables . . . . .	272
Function for getting field values . . . . .	272
Functions for storing or retrieving data for subreports . . . . .	272
Java methods available to the report design file . . . . .	273
Generating files for and running a report . . . . .	273
Exporting reports . . . . .	274
<b>Working with files and databases. . . . .</b>	<b>277</b>
SQL support. . . . .	277
EGL statements and SQL . . . . .	277
Result-set processing . . . . .	281
SQL records and their uses . . . . .	283
Database access at declaration time . . . . .	287
Dynamic SQL . . . . .	288
SQL examples . . . . .	288
Default database . . . . .	298
Informix and EGL . . . . .	299
SQL-specific tasks . . . . .	299
Retrieving SQL table data . . . . .	299
Creating dataItem parts from an SQL record part (overview). . . . .	300
Creating EGL data parts from relational database tables . . . . .	301
Viewing the SQL SELECT statement for an SQL record . . . . .	305
Validating the SQL SELECT statement for an SQL record . . . . .	305
Constructing an EGL prepare statement . . . . .	306
Constructing an explicit SQL statement from an implicit one . . . . .	306
Resetting an explicit SQL statement . . . . .	308
Removing an SQL statement from an SQL-related EGL statement. . . . .	308
Resolving a reference to display an implicit SQL statement. . . . .	308
Understanding how a standard JDBC connection is made . . . . .	309
DL/I database support . . . . .	310
EGL keywords and DL/I . . . . .	310
Record parts for DL/I support . . . . .	312
Basic DL/I database concepts . . . . .	317
Example DL/I database . . . . .	319

@DLI . . . . .	322
#dli directive . . . . .	325
Displaying implicit DL/I code. . . . .	325
DL/I-specific tasks . . . . .	326
DL/I considerations for CICS environments . . . . .	330
DL/I considerations for non-CICS environments . . . . .	333
VSAM support . . . . .	335
Access prerequisites . . . . .	335
System name . . . . .	335
MQSeries support . . . . .	336
Connections . . . . .	337
Include message in transaction . . . . .	337
Customization . . . . .	338
MQSeries-related EGL keywords . . . . .	339
Direct MQSeries calls. . . . .	341

## **IMS runtime support . . . . . 345**

EGL support for runtime PSBs and PCBs . . . . .	346
Requirements for the PSB record part . . . . .	348
Interacting with terminals in IMS. . . . .	349
Defining forms for IMS programs . . . . .	351
Estimating the size of MFS blocks for a formGroup . . . . .	351
Using serial files in IMS. . . . .	353
Using serial files as message queues. . . . .	353
Defining records to use with message queues . . . . .	355
Checking the results of serial file I/O statements . . . . .	355
Using printer files as message queues . . . . .	355
Calling an IMS program from EGL-generated Java code . . . . .	356
Example IMS program code . . . . .	358
Example of message queue I/O . . . . .	358
Example of IMS batch processing. . . . .	359
Multiple users and message queues . . . . .	359
IMS and DL/I error codes . . . . .	361

## **Maintaining EGL code . . . . . 363**

Line commenting EGL source code . . . . .	363
Searching for parts . . . . .	363
Viewing part references . . . . .	364
Viewing lists of parts. . . . .	365
Filtering lists of parts. . . . .	366
Opening a part in an .egl file . . . . .	367
Locating an EGL source file in the Project Explorer . . . . .	367
Deleting an EGL file in the Project Explorer . . . . .	368

## **Debugging EGL code . . . . . 369**

EGL debugger . . . . .	369
Debugger commands. . . . .	369
Use of build descriptors. . . . .	372
SQL-database access . . . . .	372
call statement . . . . .	373
System type used at debug time . . . . .	373
EGL debugger port . . . . .	373
Recommendations. . . . .	374
Debugging applications other than J2EE . . . . .	375
Starting a non-J2EE application in the EGL debugger. . . . .	375
Creating a launch configuration in the EGL debugger. . . . .	376

Creating an EGL Listener launch configuration	376
Debugging J2EE applications	377
Preparing a server for EGL Web debugging	377
Starting a server for EGL Web debugging	377
Starting an EGL Web debugging session	378
Using breakpoints in the EGL debugger	379
Stepping through an application in the EGL debugger	380
Viewing variables in the EGL debugger	380

## Working with EGL build parts . . . . 383

Creating a build file	383
Setting up general build options	383
Setting up external file, printer, and queue associations	393
Setting up call and transfer options	399
Setting up references to other EGL build files	406
Editing an EGL build path	407

## Generating, preparing, and running EGL output . . . . . 409

Generation	409
Generation of Java code into a project	409
Build	411
Building EGL output	413
Build plan	413
Java program, PageHandler, and library	414
Results file	414
Generating in the workbench	414
Generation in the workbench	416
Generating from the workbench batch interface	417
Generation from the workbench batch interface	417
Generating from the EGL Software Development Kit (SDK)	418
Generation from the EGL Software Development Kit (SDK)	418
Invoking a build plan after generation	419
Generating Java; miscellaneous topics	420
Processing Java code that is generated into a directory	420
Generating deployment code for EJB projects	423
Setting the variable EGL_GENERATORS_PLUGINDIR	423
Running EGL-generated Java code on the local machine	424
Starting a basic or text user interface Java application on the local machine	424
Starting a Web application on the local machine	424
Build script	426
Java build script	426
Build server	427
Starting a build server on AIX, Linux, or Windows 2000/NT/XP	427

## Deploying EGL-generated Java output 431

Java runtime properties	431
In a J2EE environment	431
In a non-J2EE Java environment	431
Build descriptors and program properties	432
For additional information	433

Setting up the non-J2EE runtime environment for EGL-generated code	433
Program properties file	433
Deploying Java applications outside of J2EE	434
Installing the EGL runtime code for Java	434
Including JAR files in the CLASSPATH of the target machine	436
Setting up the UNIX curses library for the EGL runtime	436
Setting up the TCP/IP listener for a called non-J2EE application	436
Setting up the J2EE runtime environment for EGL-generated code	437
Eliminating duplicate jar files	438
Setting deployment-descriptor values	438
Updating the J2EE environment file	440
Updating the deployment descriptor manually	441
Setting the JNDI name for EJB projects	441
Setting up the J2EE server for CICSJ2C calls	441
Setting up the J2EE server for IMSJ2C calls	442
Setting up the TCP/IP listener for a called appl in a J2EE appl client module	442
Setting up a J2EE JDBC connection	445
Deploying a linkage properties file	447
Providing access to non-EGL jar files	448

## EGL reference . . . . . 451

Assignment compatibility in EGL	451
Assignment across numeric types	452
Other cross-type assignments	452
Padding and truncation with character types	454
Assignment between timestamps	454
Assignment to or from substructured fields in fixed structures	455
Assignment of a fixed record	455
Assignments	456
Association elements	457
commit	457
conversionTable	457
fileType	457
fileName	458
formFeedOnClose	458
replace	458
system	458
systemName	459
text	459
asynchLink element	459
csouidpwd.properties file for remote calls	460
package in asynchLink element	461
recordName in asynchLink element	461
Basic record part in EGL source format	461
Build parts	462
EGL build-file format	462
Build descriptor options	464
Build scripts	498
Options required in EGL build scripts	498
callLink element	499
If callLink type is localCall (the default)	499
If callLink type is remoteCall	499
If callLink type is ejbCall	500
alias in callLink element	501



conversionTable in callLink element . . . . .	501	Enumerations in EGL. . . . .	578
ctgKeyStore in callLink element . . . . .	502	EGL reserved words . . . . .	581
ctgKeyStorePassword in callLink element . . . . .	502	Words that are reserved outside of an SQL statement. . . . .	581
ctgLocation in callLink element . . . . .	502	EGLSDK . . . . .	583
ctgPort in callLink element . . . . .	503	Syntax. . . . .	583
JavaWrapper in callLink element . . . . .	503	Examples. . . . .	584
linkType in callLink element . . . . .	504	Format of eglmaster.properties file . . . . .	585
library in callLink element . . . . .	504	EGL source format . . . . .	586
location in callLink element . . . . .	505	EGL system exceptions . . . . .	587
luwControl in callLink element . . . . .	506	EGL system limits . . . . .	590
package in callLink element . . . . .	507	Expressions . . . . .	591
parmForm in callLink element. . . . .	508	Datetime expressions . . . . .	592
pgmName in callLink element. . . . .	509	Logical expressions . . . . .	593
providerURL in callLink element . . . . .	509	Numeric expressions . . . . .	600
refreshScreen in callLink element. . . . .	510	Text expressions . . . . .	601
remoteBind in callLink element . . . . .	510	Format of master build descriptor plugin.xml file . . . . .	602
remoteComType in callLink element. . . . .	511	FormGroup part in EGL source format . . . . .	603
remotePgmType in callLink element. . . . .	513	Properties of a screen floating area . . . . .	605
serverID in callLink element . . . . .	514	Properties of a print floating area. . . . .	606
type in callLink element . . . . .	515	Form part in EGL source format . . . . .	606
C functions with EGL . . . . .	516	Text-form properties . . . . .	608
BIGINT functions for C . . . . .	519	Print-form properties . . . . .	609
C data types and EGL primitive types . . . . .	520	Form fields . . . . .	609
DATE functions for C . . . . .	521	Text-form field properties . . . . .	610
DATETIME and INTERVAL functions for C . . . . .	522	Function invocations . . . . .	613
DECIMAL functions for C . . . . .	523	Function variables. . . . .	615
Invoking a C Function from an EGL Program . . . . .	524	Function parameters . . . . .	616
Stack functions for C . . . . .	526	Implications of inOut and the related modifiers . . . . .	620
Return functions for C . . . . .	528	Function part in EGL source format . . . . .	621
CICS-related considerations . . . . .	530	Generated output . . . . .	625
Record properties across programs . . . . .	530	Generated output (reference) . . . . .	626
Temporary storage queue access on CICS . . . . .	530	Generation Results view. . . . .	627
Comments . . . . .	531	IMS-related considerations for EGL . . . . .	627
Compatibility with VisualAge Generator . . . . .	532	Transfers to and from EGL-generated IMS MPPs . . . . .	627
ConsoleUI . . . . .	533	Transfers to and from IMSADF programs . . . . .	628
ConsoleField properties and fields . . . . .	533	in operator . . . . .	629
ConsoleForm properties in EGL consoleUI. . . . .	546	Examples with a one-dimensional array . . . . .	631
Menu fields in EGL consoleUI. . . . .	547	Examples with a multidimension array. . . . .	631
MenuItem fields in EGL consoleUI . . . . .	548	Indexed record part in EGL source format. . . . .	632
PresentationAttributes fields in EGL consoleUI . . . . .	550	Interface part in EGL source format . . . . .	633
Prompt fields in EGL consoleUI . . . . .	551	Interfaces of type BasicInterface . . . . .	636
Window fields in EGL consoleUI . . . . .	553	Interfaces of type JavaObject . . . . .	637
containerContextDependent . . . . .	557	I/O error values . . . . .	638
Database authorization and table names . . . . .	557	duplicate . . . . .	639
Data conversion . . . . .	558	endOfFile. . . . .	639
Data conversion when the invoker is Java code . . . . .	559	format. . . . .	640
Conversion algorithm . . . . .	560	noRecordFound . . . . .	640
Bidirectional language text . . . . .	561	unique . . . . .	641
Data conversions between WSDL and EGL . . . . .	562	isa operator . . . . .	641
Data initialization . . . . .	564	Java runtime properties (details) . . . . .	642
DataItem part in EGL source format. . . . .	566	Java wrapper classes . . . . .	652
DataTable part in EGL source format . . . . .	568	Overview of how to use the wrapper classes . . . . .	653
EGL build path and eglpath . . . . .	571	The program wrapper class. . . . .	654
EGLCMD. . . . .	572	The set of parameter wrapper classes . . . . .	655
Syntax. . . . .	573	The set of substructured-item-array wrapper classes. . . . .	656
Examples. . . . .	574	Dynamic array wrapper classes . . . . .	657
EGL command file . . . . .	575	Naming conventions for Java wrapper classes . . . . .	659
Examples of command files . . . . .	576	Data type cross-reference . . . . .	659
EGL editor . . . . .	577	JDBC driver requirements in EGL . . . . .	660
Content assist in EGL . . . . .	577		
Source assist in EGL . . . . .	577		

Keywords . . . . .	661	Output of Java wrapper generation . . . . .	782
add . . . . .	661	Example . . . . .	783
call . . . . .	665	PageHandler part in EGL source format . . . . .	785
case . . . . .	668	PageHandler part properties . . . . .	788
close . . . . .	669	PageHandler field properties . . . . .	792
continue . . . . .	672	pfKeyEquate . . . . .	792
converse . . . . .	672	Primitive field-level properties. . . . .	793
delete . . . . .	673	@programLinkData . . . . .	797
display . . . . .	676	@xsd . . . . .	798
execute . . . . .	677	action . . . . .	800
exit . . . . .	681	alias . . . . .	801
for . . . . .	683	align . . . . .	801
forEach . . . . .	684	byPassValidation . . . . .	802
forward . . . . .	686	color . . . . .	802
freeSQL . . . . .	687	column . . . . .	803
get . . . . .	687	currency . . . . .	804
get absolute . . . . .	695	currencySymbol . . . . .	805
get current . . . . .	697	dateFormat . . . . .	805
get first . . . . .	698	displayName . . . . .	807
get last . . . . .	699	displayNames . . . . .	808
get next . . . . .	701	displayUse . . . . .	808
get next inParent . . . . .	707	dliFieldName . . . . .	809
get previous . . . . .	708	fieldLen . . . . .	809
get relative . . . . .	713	fill . . . . .	810
goTo . . . . .	714	fillCharacter . . . . .	810
if, else . . . . .	715	help . . . . .	810
move . . . . .	716	highlight . . . . .	811
open . . . . .	722	inputRequired . . . . .	811
openUI . . . . .	726	inputRequiredMsgKey . . . . .	811
prepare . . . . .	736	intensity . . . . .	812
print . . . . .	737	isBoolean . . . . .	812
replace . . . . .	738	isDecimalDigit . . . . .	813
return . . . . .	741	isHexDigit . . . . .	813
set . . . . .	742	isNullable . . . . .	813
show . . . . .	751	isReadOnly . . . . .	814
transfer . . . . .	752	lineWrap . . . . .	815
try . . . . .	754	lowerCase . . . . .	815
while . . . . .	755	masked . . . . .	816
Library (generated output) . . . . .	756	maxLen . . . . .	816
Library part in EGL source format . . . . .	756	minimumInput . . . . .	816
like operator. . . . .	763	minimumInputMsgKey . . . . .	817
Linkage properties file (details) . . . . .	764	modified . . . . .	817
How the linkage properties file is identified at		needsSOSI . . . . .	818
run time . . . . .	764	newWindow . . . . .	818
Format of the linkage properties file. . . . .	764	numElementsItem . . . . .	819
matches operator . . . . .	766	numericFormat . . . . .	819
Message customization for EGL Java run time . . . . .	768	numericSeparator . . . . .	820
MQ record part in EGL source format . . . . .	769	runValidatorFromProgram . . . . .	820
MQ record properties. . . . .	772	outline . . . . .	820
Queue name. . . . .	772	pattern . . . . .	821
Include message in transaction . . . . .	772	persistent. . . . .	821
Open input queue for exclusive use . . . . .	772	protect . . . . .	822
Options records for MQ records . . . . .	772	selectedIndexItem . . . . .	823
Name aliasing . . . . .	774	selectFromListItem . . . . .	823
Changes to EGL identifiers in JSP files and		selectType . . . . .	824
generated Java beans . . . . .	774	sign . . . . .	825
How Java names are aliased . . . . .	776	sqlDataCode. . . . .	825
How Java wrapper names are aliased . . . . .	776	sqlVariableLen . . . . .	826
Naming conventions . . . . .	778	timeFormat . . . . .	827
Operators and precedence . . . . .	779	timestampFormat . . . . .	828
Output of Java program generation . . . . .	781	typeChkMsgKey . . . . .	829

uiType . . . . .	829
upperCase . . . . .	831
validationOrder . . . . .	831
validatorDataTable . . . . .	832
validatorDataTableMsgKey . . . . .	833
validatorFunction . . . . .	833
validatorFunctionMsgKey . . . . .	834
validValues . . . . .	834
validValuesMsgKey . . . . .	836
value . . . . .	836
zeroFormat . . . . .	837
Program data other than parameters . . . . .	837
Program parameters . . . . .	840
Program part in EGL source format . . . . .	841
Basic program in EGL source format . . . . .	842
Text UI program in EGL source format . . . . .	844
VGWebTransaction program in EGL source format. . . . .	847
Program part properties . . . . .	856
Input form . . . . .	859
Input record. . . . .	859
Record and file type cross-reference . . . . .	860
Properties that support variable-length records . . . . .	860
Variable-length records with the lengthItem property . . . . .	860
Variable-length records with the numElementsItem property. . . . .	861
Variable-length records with both lengthItem and numElementsItem properties. . . . .	862
Variable-length records passed on a call or transfer . . . . .	862
Reference compatibility in EGL . . . . .	862
For reference variables . . . . .	862
For non-reference variables. . . . .	863
Reference variables and NIL in EGL. . . . .	863
Relative record part in EGL source format. . . . .	865
Run unit . . . . .	866
resultSetID . . . . .	867
Serial record part in EGL source format . . . . .	868
Service part in EGL source format . . . . .	869
SQL data codes and EGL host variables . . . . .	874
Variable and fixed-length columns . . . . .	874
Compatibility of SQL data types and EGL primitive types . . . . .	875
VARCHAR, VARGRAPHIC, and the related LONG data types . . . . .	876
DATE, TIME, and TIMESTAMP . . . . .	876
SQL record internals . . . . .	876
SQL record part in EGL source format . . . . .	877
Structure field in EGL source format. . . . .	880
Substrings . . . . .	882
Syntax diagram for EGL functions . . . . .	884
Syntax diagram for EGL statements and commands . . . . .	884
System Libraries . . . . .	886
EGL library ConsoleLib . . . . .	886
EGL library ConverseLib . . . . .	916
EGL library DateTimeLib . . . . .	919
EGL library DLILib . . . . .	929
EGL library J2EELib . . . . .	931
EGL library JavaLib . . . . .	934
EGL library LobLib . . . . .	958

EGL library MathLib . . . . .	966
recordName.resourceAssociation . . . . .	985
EGL library ServiceLib . . . . .	986
EGL library ReportLib . . . . .	990
EGL library StrLib. . . . .	996
EGL library SysLib . . . . .	1016
EGL library VGLib . . . . .	1047
System variables outside of EGL libraries. . . . .	1056
ConverseVar . . . . .	1057
DLIVar . . . . .	1062
SysVar . . . . .	1063
VGVar . . . . .	1077
transferToTransaction element . . . . .	1088
alias in transfer-related linkage elements . . . . .	1089
externallyDefined in transferToTransaction element . . . . .	1089
VGUIRecord part in EGL source format . . . . .	1089
Use declaration . . . . .	1091
Background . . . . .	1091
In a program or library part . . . . .	1092
In a formGroup part . . . . .	1094
In a pageHandler part . . . . .	1095

## EGL Java runtime error codes . . . . 1097

EGL Java runtime error code CSO7000E . . . . .	1098
EGL Java runtime error code CSO7015E . . . . .	1099
EGL Java runtime error code CSO7016E . . . . .	1099
EGL Java runtime error code CSO7020E . . . . .	1099
EGL Java runtime error code CSO7021E . . . . .	1100
EGL Java runtime error code CSO7022E . . . . .	1100
EGL Java runtime error code CSO7023E . . . . .	1100
EGL Java runtime error code CSO7024E . . . . .	1100
EGL Java runtime error code CSO7026E . . . . .	1100
EGL Java runtime error code CSO7045E . . . . .	1101
EGL Java runtime error code CSO7050E . . . . .	1101
EGL Java runtime error code CSO7060E . . . . .	1101
EGL Java runtime error code CSO7080E . . . . .	1101
EGL Java runtime error code CSO7160E . . . . .	1102
EGL Java runtime error code CSO7161E . . . . .	1102
EGL Java runtime error code CSO7162E . . . . .	1102
EGL Java runtime error code CSO7163E . . . . .	1103
EGL Java runtime error code CSO7164E . . . . .	1103
EGL Java runtime error code CSO7165E . . . . .	1103
EGL Java runtime error code CSO7166E . . . . .	1103
EGL Java runtime error code CSO7360E . . . . .	1104
EGL Java runtime error code CSO7361E . . . . .	1104
EGL Java runtime error code CSO7488E . . . . .	1104
EGL Java runtime error code CSO7489E . . . . .	1105
EGL Java runtime error code CSO7610E . . . . .	1105
EGL Java runtime error code CSO7620E . . . . .	1105
EGL Java runtime error code CSO7630E . . . . .	1106
EGL Java runtime error code CSO7640E . . . . .	1106
EGL Java runtime error code CSO7650E . . . . .	1106
EGL Java runtime error code CSO7651E . . . . .	1107
EGL Java runtime error code CSO7652E . . . . .	1107
EGL Java runtime error code CSO7653E . . . . .	1108
EGL Java runtime error code CSO7654E . . . . .	1108
EGL Java runtime error code CSO7655E . . . . .	1109
EGL Java runtime error code CSO7656E . . . . .	1109
EGL Java runtime error code CSO7657E . . . . .	1110
EGL Java runtime error code CSO7658E . . . . .	1111

**x** EGL Reference Guide





xii EGL Reference Guide

EGL Java runtime error code VGJ1416E . . . .	1213
EGL Java runtime error code VGJ1417E . . . .	1213
EGL Java runtime error code VGJ1419E . . . .	1213
EGL Java runtime error code VGJ1501E . . . .	1213
EGL Java runtime error code VGJ1502E . . . .	1214
EGL Java runtime error code VGJ1503E . . . .	1214
EGL Java runtime error code VGJ1504E . . . .	1214
EGL Java runtime error code VGJ1525E . . . .	1214
EGL Java runtime error code VGJ1526E . . . .	1215
EGL Java runtime error code VGJ1527E . . . .	1215
EGL Java runtime error code VGJ1528E . . . .	1215
EGL Java runtime error code VGJ1529E . . . .	1215
EGL Java runtime error code VGJ1530E . . . .	1216
EGL Java runtime error code VGJ1532E . . . .	1216
EGL Java runtime error code VGJ1534E . . . .	1216
EGL Java runtime error code VGJ1535E . . . .	1216
EGL Java runtime error code VGJ1536E . . . .	1217
EGL Java runtime error code VGJ1537E . . . .	1217

EGL Java runtime error code VGJ1538E . . . .	1217
EGL Java runtime error code VGJ1539E . . . .	1218
EGL Java runtime error code VGJ1540E . . . .	1218
EGL Java runtime error code VGJ1541E . . . .	1218
EGL Java runtime error code VGJ1542E . . . .	1219
EGL Java runtime error code VGJ1543E . . . .	1219
EGL Java runtime error code VGJ1544E . . . .	1219
EGL Java runtime error code VGJ1545E . . . .	1219
EGL Java runtime error code VGJ9900E . . . .	1220
EGL Java runtime error code VGJ9901E . . . .	1220

## **Appendix. Notices . . . . . 1223**

Programming interface information . . . . .	1225
Trademarks and service marks . . . . .	1225

## **Index . . . . . 1227**



---

## Overview

---

### Introduction to EGL

Enterprise Generation Language (EGL) is a development environment and programming language that lets you write full-function applications quickly, freeing you to focus on the business problem your code is addressing rather than on software technologies. You can use similar I/O statements to access different types of external data stores, for example, whether those data stores are files, relational databases, or message queues. The details of Java™ and J2EE are hidden from you, too, so you can deliver enterprise data to browsers even if you have minimal experience with Web technologies.

After you code an EGL program, you generate it to create Java source; then EGL prepares the output to produce executable objects. EGL also can provide these services:

- Places the source on a deployment platform outside of the development platform
- Prepares the source on the deployment platform
- Sends status information from the deployment platform to the development platform, so you can check the results

EGL even produces output that facilitates the final deployment of the executable objects.

An EGL program written for one target platform can be converted easily for use on another. The benefit is that you can code in response to current platform requirements, and many details of any future migration are handled for you. EGL also can produce multiple parts of an application system from the same source.

#### **Related concepts**

"Development process" on page 9

"EGL projects, packages, and files" on page 15

"Generated output" on page 625

"Parts" on page 19

"Runtime configurations" on page 11

#### **Related tasks**

"Creating an EGL Web project" on page 127

"Improving performance at development time" on page 10

#### **Related reference**

"EGL editor" on page 577

"EGL source format" on page 586

---

### What's new in EGL 6.0.1

Version 6.0.1 includes the following changes:

- You can now use EGL syntax to define a service, which is a set of functions that are accessible by other application components. You have a choice of deploying the service in either of these ways:

- As a *Web service*, which can be accessed from any code by way of an HTTP connection
- As an *EGL service*, which can be accessed from EGL code directly or by way of a TCP/IP connection
- You can access external code from within EGL code--
  - *EGL interface technology* lets you access various types of code with the syntax used to access a library function. You can access an EGL service, a Web service (whether or not written with EGL), or Java code.  
 At the center of this technology is the Interface part, which you can code by hand. In addition, new wizards create Interface parts either from an EGL service or from a Web Service Description Language (WSDL) file.
  - EGL lets you dynamically update aspects of JSF controls that are displayed at a Web browser. In response to a user's input you can change the color of a text box or can add or remove controls, for example. The changes occur on the Web application server, affecting the information available to the JSP that in turn presents the Web page.  
 A new source assistant helps you to code those dynamic changes. First, the assistant presents a hierarchical list of the JSF controls in your JSP file. Second, after you select one of the controls, the assistant places control-specific code into your source file.
- Other aspects of the user interface are new--
  - The EGL Parts List view displays parts in an on-screen table that can be sorted by name, type of part, location, filename, and package. From this view, you can perform several different operations on parts, such as generating them, viewing a subset of their references and declarations, and opening them in the EGL editor or in the EGL Parts Reference view.
  - A new source assistant helps you to create DataItem parts, prompting you for property values and validating your choices.
- A new program type is available (VGWebTransaction), primarily to support the migration of Web transactions from VisualAge® Generator.
- EGL now lets you develop code that accesses the IMS™ runtime platform and DL/I databases. To generate that code, you need EGL COBOL Generation feature, which is available only with WebSphere® Developer for zSeries®.
- EGL also lets you call an IMS program from an EGL-generated Java program.
- The new consoleUI function **updateWindowAttributes** refreshes aspects of the active consoleUI window.
- Various additions were made to the EGL syntax, including these--
  - The literal value NIL is now available to indicate that a reference variable does not refer to a value.
  - EGL introduces complex properties, each composed of a set of property fields. You might use syntax like the following, for example, to declare an EGL service and to define the complex property @EGLBinding, which contains the details needed to provide access to the service:
 

```

myService myServicePart
{
  @EGLBinding
  {
    commType="DIRECT",
    serviceName="myService",
    servicePackage="my.useful.service"
  }
}
          
```
  - The rules for setting initializers in fixed-record parts are specified; for instance, only a leaf field can be assigned an initial value.



- You can now use the function parameter modifiers **IN** and **OUT** with a record that has no fixed structure, unless the record is used in an I/O statement. (The modifier **INOUT** was already supported for records with or without a fixed structure.)

#### Related tasks

“Migrating EGL code to EGL V6.0.1” on page 101

---

## What’s new in EGL 6.0.0.1

Version 6.0.0.1 includes the following changes:

- The EGL form editor provides a graphical user interface for creating text and print forms.
- Target environments include HP-UX and Solaris. EGL provides 32- and 64-bit support for those platforms and has added 64-bit support for AIX®.
- The EGL debugger has the following changes:
  - Allows you to debug consoleUI-based applications
  - Allows use of an EBCDIC code page to represent character and numeric data during a debugging session
- The language is more flexible:
  - The system variables **SysVar.sqlCode** and **SysVar.sqlState** are modifiable
  - Array subscripts and substring indexes can include numeric expressions, as long as those expressions do not include functions
  - Any function that returns a value can be invoked from within a numeric, text, or logical expression, if the type of the return value is valid in the expression
  - Any function that returns a value can be used as an argument to a function parameter that has the modifier **in**, if the return value and parameter types are assignment compatible
  - Any EGL system variable can be passed as an argument to any function parameter that has the modifier **in**, if the argument and parameter types are assignment compatible
  - Any modifiable EGL system variable can be passed as an argument to a function parameter that has the modifier **out** (if the argument and parameter types are assignment compatible) or **inOut** (if the argument and parameter types are reference compatible)
- Documentation now identifies the access modifier (**in**, **out**, or **inOut**) for every parameter in every EGL system function; and describes reference and assignment compatibility
- New system functions are available:
  - **MathLib.stringAsDecimal** accepts a character value (like "98.6") and returns the equivalent value of type DECIMAL.
  - **MathLib.stringAsFloat** accepts a character value (like "98.6") and returns the equivalent value of type FLOAT.
  - **MathLib.stringAsInt** accepts a character value (like "98") and returns the equivalent value of type BIGINT.
  - **SysLib.conditionAsInt** accepts a logical expression (like *myVar == 6*) , returning a 1 if the expression is true, a 0 if the expression is false.
  - **SysLib.startLog** opens an error log. Text is written into that log every time your program invokes **SysLib.errorLog**.

- **SysLib.errorLog** copies text into the error log that was started by the system function **SysLib.startLog**
- New functions support consoleUI--
  - **ConsoleLib.currentArrayCount** returns the number of elements in the dynamic array that is associated with the current active form
  - **ConsoleLib.setCurrentArrayCount** specifies how many rows exist in a dynamic array that is bound to an on-screen arrayDictionary
  - **ConsoleLib.hideAllMenuItems** hides all menuItems in the currently displayed menu
  - **ConsoleLib.showAllMenuItems** shows all menuItems in the currently displayed menu
- The Informix® 4GL conversion tool is included with the product
- The VAGen migration tool has changes that allow for a more efficient migration

#### Related concepts

“Sources of additional information on EGL” on page 14

---

## What’s new in the EGL 6.0 iFix

**Note:** EGL provides services to help you convert old code to code that works with the EGL 6.0 iFix:

- If you used a pre-6.0 version of EGL to create a Web application that is based on JavaServer Faces, do as follows in the workbench--
  1. Click **Help > Rational Help**
  2. In the **Search** text box of the help system, type at least the initial characters in this string: *Migrating JavaServer Faces resources in a Web project*
  3. Click **GO**
  4. Click *Migrating JavaServer Faces resources in a Web project* and follow the directions in that topic
- For other details on migrating code from EGL 6.0 or from an earlier version, see *Migrating EGL code to the EGL 6.0 iFix*.
- If you are migrating code from Informix 4GL or from VisualAge Generator, see *Sources of additional information on EGL*.

The version 6.0 iFix represents a significant upgrade to the EGL language:

- Introduces the EGL report handler, which contains customized functions that are invoked at different times during execution of a JasperReports design file. The data returned from each function is included in your output report, which can be rendered in PDF, XML, text, or HTML format. The technology is an improvement on the reporting capability that was available in Informix 4GL.
- Introduces the EGL console UI, which is a technology for creating a character-based interface that allows an immediate, keystroke-driven interaction between the user and an EGL-generated Java program. The technology is an improvement on the dynamic user interface that was available in Informix 4GL.
- Provides new flexibility for code development--
  - Allows you to declare new types of variables:
    - A reference variable, which does not contain business data but points to such data.



- A variable that contains or refers to a large quantity of data; specifically, to a binary large object (BLOB) or a character large object (CLOB).
- A string variable, which refers to a Unicode string whose length varies at run time.
- An ANY-typed variable, which can contain business data of any primitive type.
- Allows you to include function invocations in expressions.
- Allows you to reference a record without having development-time knowledge of the size or other characteristics of the record or of the fields in that record. Each field can itself refer to a record.
- Expands support for dynamic arrays, which can now have multiple dimensions.
- Introduces two new kinds of data collections:
  - A dictionary, which is composed of a set of key-and-value entries. You can add, delete, and retrieve entries at runtime, and the value in a given entry can be of any type.
  - An arrayDictionary, which is composed of a set of one-dimensional arrays, each of any type. You access the content of an arrayDictionary by retrieving the same-numbered elements across all the arrays.
- Expands the number of system functions for various purposes:
  - To improve datetime processing, runtime message handling, and retrieval of user-defined Java runtime properties.
  - To support the new functionality related to reports, console UI, BLOB, and CLOB.
- Provides better support for exception handling, for data initialization, and for DLL access.
- Provides a new wizard to create EGL report handlers.
- Allows you to customize a Web-page template for use with the Data Parts and Pages wizard, which quickly provides a Web application for accessing a single relational database.
- Allows you to create code that reflects the runtime behavior of Informix 4GL in relation to null processing and database commits.

#### Related concepts

"EGL-to-EGL migration" on page 104

"Sources of additional information on EGL" on page 14

"What's new in EGL version 6.0"

---

## What's new in EGL version 6.0

**Note:** If you used an earlier version of EGL to create a Web application that is based on JavaServer Faces, do as follows in the workbench:

1. Click **Help > Help Contents**
2. In the **Search** text box of the help system, type at least the initial characters in this string: *Migrating JavaServer Faces resources in a Web project*
3. Click **GO**
4. Click *Migrating JavaServer Faces resources in a Web project* and follow the directions in that topic

Version 6.0 increases the power of the EGL language:

- Processing of relational databases has improved--
  - New wizards let you quickly do as follows:
    - Create data parts directly from relational database tables
    - Create Web applications that create, read, update, and delete table rows from such tables
  - New system functions are available:
    - **sysLib.loadTable** loads information from a file and inserts it into a relational database table.
    - **sysLib.unloadTable** unloads information from a relational database table and inserts it into a file.
  - If you are generating Java code, you can access SQL database rows in a cursor by navigating to the next row (as was always true); by navigating to the first, last, previous, or current row; or by specifying an absolute or relative position in the cursor.
  - The **forEach** statement allows you to loop easily through the rows of an SQL result set.
  - The **freeSQL** statement frees any resources associated with a dynamically prepared SQL statement, closing any open cursor associated with that SQL statement.
- String processing has improved--
  - You can specify substrings in a text expression, as in the following example:
 

```
myItem01 = "1234567890";

// myItem02 = "567"
myItem02 = myItem01[5:7];
```
  - You can specify a back space, form feed, or tab in a text literal
  - You can compare strings against either of two pattern types:
    - An SQL-type pattern, which includes the LIKE keyword. An example is as follows:
 

```
// variable myVar01 is the string expression
// whose contents will be compared to a like criterion
myVar01 = "abcdef";

// the next logical expression evaluates to "true"
if (myVar01 like "a_c%")
;
end
```
    - A regular-expression pattern. An example is as follows:
 

```
// variable myVar01 is the string expression
// whose contents will be compared to a match criterion
myVar01 = "abcdef";

// the next logical expression evaluates to "true"

if (myVar01 matches "a?c*")
;
end
```
  - You can use these text-formatting system functions:
    - strLib.characterAsInt**  
Converts a character string into an integer string
    - strLib.clip**  
Deletes trailing blank spaces and nulls from the end of returned character strings

**strLib.formatNumber**

Returns a number as a formatted string

**strLib.integerAsChar**

Converts an integer string into a character string

**strLib.lowercase**

Converts all uppercase values in a character string to lowercase values

**strLib.spaces**

Returns a string of a specified length.

**strLib.upperCase**

Converts all lowercase values in a character string to uppercase values.

- You can declare variables and structure items of new types.

The new numeric types are as follows--

**FLOAT**

Concerns an 8-byte area that stores a double-precision floating-point numbers with as many as 16 significant digits

**MONEY**

Concerns a currency amount that is stored as a fixed-point decimal number up to as many as 32 significant digits

**SMALLFLOAT**

Concerns a 4-byte area that stores a single-precision floating-point number with as many as 8 significant digits

The new datetime types are as follows--

**DATE**

Concerns a specific calendar date, as represented in 8 single-byte digits

**INTERVAL**

Concerns a span of time that is represented in 1 to 21 single-byte digits and is associated with a mask such as "hhmmss" for hours, minutes, and seconds

**TIME**

Concerns an instance in time, as represented in 6 single-byte digits

**TIMESTAMP**

Concerns an instance in time that is represented in 1 to 20 single-byte digits and is associated with a mask such as "yyyyMMddhh" for year, month, day, and hour

- The syntax provides additional options--

- You could always reference an element of a structure-item array as follows, but in light of iFix changes, you are asked to avoid this syntax:

```
mySuperItem.mySubItem.mySubmostItem[4,3,1]
```

The following syntax is strongly recommended--

```
mySuperItem[4].mySubItem[3].mySubmostItem[1]
```

- You can use a comma-delineated list of identifiers when you declare parameters, use-statement entries, set-statement entries, or variables, as in this example:

```
myVariable01, myVariable02 myPart;
```

- In a numeric expression, you can now specify an exponent by preceding a value with a double asterisk (\*\*), so that (for example) 8 cubed is 8\*\*3

- You can now specify expressions that each resolve to a date, time, timestamp, or interval; and date arithmetic lets you do various tasks such as calculating the number of minutes between two dates
- The following additions also allow for date and time processing:
  - **DateTimeLib.currentTime** and **DateTimeLib.currentTimeStamp** are system variables that reflect the current time
  - New formatting functions are available for dates (**StrLib.formatDate**), times (**StrLib.formatTime**), and timestamps (**sysLib.TimeStamp**)
  - Each of the following functions let you convert a series of characters to an item of a datetime type so that the item can be used in a datetime expression:
    - **DateTimeLib.dateValue** returns a date
    - **DateTimeLib.timeValue** returns a time
    - **DateTimeLib.timestampValue** returns a timestamp associated with a particular mask such as "yyyyMMdd"
    - **DateTimeLib.intervalValue** returns an interval associated with a particular mask such as "yyyyMMdd"
    - **DateTimeLib.extendDateTimeValue** accepts a date, time, or timestamp and extends it to an item associated with a particular mask such as "yyyyMMddmmss"
- You can use these new, general statements:
  - The **for** statement includes a statement block that runs in a loop for as many times as a test evaluates to true. The test is conducted at the beginning of the loop and indicates whether the value of a counter is within a specified range.
  - The **continue** statement transfers control to the end of a **for**, **forEach**, or **while** statement that itself contains the **continue** statement. Execution of the containing statement continues or ends depending on the logical test that is conducted at the start of the containing statement.
- You can run a system command synchronously (by issuing the function **sysLib.callCmd**) or asynchronously (by issuing the function **sysLib.startCmd**).
- You can use two new functions that let you access command-line arguments in a loop
  - **sysLib.callCmdLineArgCount** returns the number of arguments
  - **sysLib.callCmdLineArg** returns the argument that resides in a specified position in the list of arguments
- You can now specify a **case** statement in which each clause is associated with a different logical expression. If you use this new syntax, the EGL runtime executes the statements that are associated with the first true expression:
 

```

case
  when (myVar01 == myVar02)
    conclusion = "okay";
  when (myVar01 == myVar03)
    conclusion = "need to investigate";
  otherwise
    conclusion = "not okay";
end
      
```
- You can control whether a function parameter is used only for input, only for output, or for both; and you can avoid the choice by accepting the default setting, which is the unrestricted "for both".
- You can now specify a datetime, text, or numeric expression that is more complex than a single item or constant, in these cases:

- When you specify the value that is provided to the operating system by a **return** statement
- When you specify an argument that is passed in either a function invocation or a program call; however, the characteristics of the receiving parameter must be known at generation time
- You can now specify a complex numeric expression when exiting from the program

The development environment has improved as well:

- Two new features give you the ability to access parts quickly, even as your code grows in complexity--
  - The Parts Reference view lets you display a hierarchical list of the EGL parts that are referenced by a program, library, or PageHandler; and from that list, you can access any of the referenced parts
  - The EGL search mechanism lets you specify a search criterion to access a set of parts or variables in your workspace or in a subset of your projects
- Finally, the EGL Web perspective has been eliminated in favor of the widely used Web perspective.

#### **Related concepts**

"EGL-to-EGL migration" on page 104

"Sources of additional information on EGL" on page 14

"What's new in the EGL 6.0 iFix" on page 4

---

## **Development process**

Your work with EGL includes the following steps:

### **Setup**

You set up a work environment; for example, you set preferences and create projects.

### **Create and open EGL source files**

You begin to create the source code.

### **Declaration**

You create and specify the details of your code.

### **Validation**

At various times (such as when you save a file), EGL reviews your declarations and indicates whether they are syntactically correct and (to some extent) whether they are internally consistent.

### **Debugging**

You can interact with a built-in debugger to ensure that your code fulfills your requirements.

### **Generation**

EGL validates your declarations and creates output, including source code.

### **Preparation**

EGL prepares the source code to produce executable objects. In some cases this step places the source code on a deployment platform outside of the development platform, prepares the source code on the deployment platform, and sends a results file from the deployment platform to the development platform.

### Run through

In some cases, you can run your code immediately in the Workbench just by right-clicking the Java output and clicking **Run > Java application**.

### Deployment

EGL produces output that makes deployment of the executable objects easier.

### Related concepts

"EGL debugger" on page 369

"Generation of Java code into a project" on page 409

"Introduction to EGL" on page 1

### Related tasks

"Processing Java code that is generated into a directory" on page 420

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

### Related reference

"EGL editor" on page 577

"EGL source format" on page 586

---

## Troubleshooting

### Out of date .jar files

#### Symptom

Unpredictable; generated code might not compile, generated code might not run correctly, or the debugger might not work properly.

#### Cause

EGL compares the timestamp (last time modified) of the .jar files in its plugin with the timestamp of the .jar files in your project. If the files in your project are older, EGL replaces them. Sometimes an external tool will update the timestamps of your files, tricking the generator into thinking that your outdated files don't need to be replaced.

#### Solution

Delete the Jar files from your project and generate code from an EGL source file. After that you can be assured that you have the latest files.

#### Related concepts

#### Related tasks

#### Related reference

---

## Improving performance at development time

If you converted code from Informix 4GL or if you are creating applications that include many interdependencies (as when library functions invoke functions in other libraries), you may get better performance at development time if you re-configure the Java Virtual Machine that supports your IBM® product:

1. In Windows® Explorer, navigate to the installation directory that contains the file `rationalsdpi.ini`. The directory may be as follows:  
C:\Program Files\IBM\Rational\SDP\6.0
2. Open the file `rationalsdpi.ini` in a text editor
3. Add the following **on a single line** after the last line that begins with `VMargs`:

**Related concepts**

“Introduction to EGL” on page 1

“Sources of additional information on EGL” on page 14

---

## Runtime configurations

EGL allows you to generate a Java program for any of several supported platforms. You can deploy the program outside of J2EE or in the context of any of the following J2EE containers--

- J2EE application client
- J2EE Web application
- EJB container; in this case, you also generate an EJB session bean

You can call an EGL-generated Java program in an EGL service or EGL Web service, but only if the program is non-interactive. The program in this case may be deployed inside or outside of J2EE.

In addition, EGL provides a way to define a Web application that has the following characteristics:

- Delivers graphical pages to Web browsers
- Is able to store and retrieve data for a potentially large number of users
- Is embedded in a Java-based framework called JavaServer Faces

For details on this specialized support for Web applications, see *PageHandler part*.

You also can update a program of type VGWebTransaction, although the best practice is to use PageHandler parts for developing Web applications. EGL includes the program type to support migration of a VisualAge Generator program type that allowed developers to structure a Web application as if the code were running in a non-Web environment. The program of type VGWebTransaction retrieves data, displays data, accepts user input, and post-processes data, whereas most applications built with PageHandler parts forward control from one Web page to the next.

A program of type VGWebTransaction is a main program (not called) and runs outside of J2EE.

Finally, you can use EGL to generate a Java wrapper, as described in the next section.

## Use of a Java wrapper

The EGL-generated Java wrapper is a set of classes that lets you invoke an EGL-generated program from non-EGL-generated Java code; for example, from an action class in a Struts- or JSF-based J2EE web application or from a non-J2EE Java program. The Java-to-EGL integration task is as follows:

1. Generate Java wrapper classes, which are specific to a generated program
2. Incorporate those wrapper classes into the non-generated Java code
3. From the non-generated Java code, invoke the wrapper-class methods to make the actual call and to convert data between these two formats:
  - The data-type formats used by Java

- The primitive-type formats required when passing data to and from the EGL-generated program

## Valid calls

The next table shows the valid calls to or from the EGL-generated code.

Calling object	Called object
An EGL-generated Java wrapper in a Java class that is outside of J2EE	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	An IMS/VS program (generated by EGL or not)
	A CICS <sup>®</sup> program (generated by EGL or not)
An EGL-generated Java wrapper in a J2EE application client	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	An IMS/VS program (generated by EGL or not)
	A CICS program (generated by EGL or not)
An EGL-generated Java wrapper in a J2EE Web application	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated Java program in the same J2EE Web application
	An EGL-generated EJB session bean
	An IMS/VS program (generated by EGL or not)
	A CICS program (generated by EGL or not)
An EGL-generated Java program that is outside of J2EE	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A non-EGL-generated program that was written in C or C++
	An IMS/VS program (generated by EGL or not)
	A CICS program (generated by EGL or not)



Calling object	Called object
An EGL-generated Java program that is in a J2EE application client	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A non-generated program that was written in C or C++
	An IMS/VS program (generated by EGL or not)
	A CICS program (generated by EGL or not)
An EGL-generated Java program in a J2EE Web application	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated Java program in the same J2EE Web application
	An EGL-generated EJB session bean
	A non-generated program written in C or C++
	An IMS/VS program (generated by EGL or not)
An EGL-generated EJB session bean	A CICS program (generated by EGL or not)
	An EGL-generated Java program (non-J2EE)
	An EGL-generated Java program in a J2EE application client
	An EGL-generated EJB session bean
	A non-generated program written in C or C++
	An IMS/VS program (generated by EGL or not)
	A CICS program (generated by EGL or not)

## Valid transfers

The next table shows the valid transfers to or from EGL-generated code.

Transferring object	Receiving object
An EGL-generated Java program that is outside of J2EE	An EGL-generated Java program (non-J2EE)
An EGL-generated Java program that is in a J2EE application client	An EGL-generated Java program in the same J2EE application client
An EGL-generated Java program in a J2EE Web application	An EGL-generated Java program in the same J2EE Web application

## Related concepts

“Generated output” on page 625

“Introduction to EGL” on page 1

"Java program, PageHandler, and library" on page 414

"Java wrapper" on page 390

"PageHandler" on page 223

#### **Related tasks**

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

---

## **Sources of additional information on EGL**

The most recent copy of this document is at the following Web site:

<http://www.ibm.com/developerworks/rational/library/egldoc.html>

For details on migrating source code written in VisualAge Generator, see the *VisualAge Generator to EGL Migration Guide* (file `vagenmig.pdf`), which is on the Web site mentioned earlier and in the help system section called *Installing and migrating*.)

However, it is recommended that you access the Web site mentioned earlier.

However, it is recommended that you access the Web site.

#### **Related concepts**

"Introduction to EGL" on page 1

---

## EGL language overview

---

### EGL projects, packages, and files

An EGL project includes zero to many source folders, each of which includes zero to many packages, each of which includes zero to many files. Each file contains zero to many parts.

#### EGL project

An EGL project is characterized by a set of properties, which are described later. In the context of an EGL project, EGL automatically performs validation and resolves part references when you perform certain tasks; for example, when you save an EGL source file or build file. In addition, if you are working with PageHandler parts or VGUIRecords, EGL automatically generates output, but only in this case:

- You have set the automatic build process after selecting these options: **Window > Preferences > Workbench > Perform build automatically on resource modification**
- You have established a default build descriptor as a preference or property

An EGL project is formed by selecting **EGL** or **EGL Web** as the project type when you create a new project. You assign properties while working through the steps of project creation. To begin modifying your choices after you have completed those steps, right-click the project name and when a context menu is displayed, click **Properties**.

The EGL properties are as follows:

##### EGL source folder

One or more project folders that are the roots for the project's packages, each of which is a set of subdirectories. A source folder is useful for keeping EGL source separate from Java files and for keeping EGL source files out of the Web deployment directories. It is recommended that you specify EGL source folders in all cases; but if a source folder is not specified, the only source folder is the project directory.

The value of this property is stored in a file named `.eglp` in the project directory and is saved in the repository (if any) that you use to store EGL source files.

The EGL project wizards each create one source folder named **EGLSource**.

##### EGL build path

The list of projects that are searched for any part that is not found in the current project.

The value of this property is stored in a file named `.eglp` in the project directory and is saved in the repository (if any) that you use to store EGL source files.

In the following example of an `.eglp` file, `EGLSource` is a source folder in the current project, and `AnotherProject` is a project in the EGL path:

```
<?xml version="1.0" encoding="UTF-8"?>
<eglp>
  <eglpentry kind="src" path="EGLSource"/>
  <eglpentry kind="src" path="\AnotherProject"/>
</eglp>
```

The source folders for AnotherProject are determined from the .eglpth file in *that* project.

#### Default build descriptors

The build descriptors that allow you to generate output quickly, as described in *Generation in the workbench*.

## Package

A package is a named collection of related source parts. The name is case sensitive: *myPkg* is different from *MYPKG*.

No package is in use when you create build parts.

By convention, you achieve uniqueness in package names by making the initial part of the package name an inversion of your organization's Internet domain name. For example, the IBM domain name is *ibm.com*<sup>®</sup>, and the EGL packages begin with "com.ibm". By using this convention, you gain some assurance that the names of Web programs developed by your organization will not duplicate the names of programs developed by another organization and can be installed on the same server without possibility of a name collision.

The folders of a given package are identified by the package name, which is a sequence of identifiers separated by periods (.), as in this example:

`com.mycom.mypack`

Each identifier corresponds to a subfolder under an EGL source folder. The directory structure for `com.mycom.mypack`, for example, is `\com\mycom\mypack`, and the source files are stored in the bottom-most folder; in this case, in `mypack`. If the workspace is `c:\myWorkspace`, if the project is *new.project*, and if the source folder is `EGLSource`, the path for that package is as follows:

`c:\myWorkspace\new.project\EGLSource\com\mycom\mypack`

The parts in an EGL source file all belong to the same package. The file's package statement, if any, specifies the name of that package. If you do not specify a package statement, the parts are stored directly in the source folder and are said to be in the *default package*. It is recommended that you always specify a package statement because files in the default package cannot be shared by parts in other packages or projects.

Two parts with the same identifier may not be defined in the same package. *It is strongly recommended that you avoid using the same package name under different projects or different folders.*

The package for generated Java output is the same as the EGL source file package.

## EGL files

Each EGL file belongs to one of these categories:

#### Source file

An EGL source file (extension .egl) contains logic, data, and user interface parts and is written in EGL source format.

Each of the following *generatable parts* can be transformed into a compilable unit:

- DataTable

- FormGroup
- Handler (the basis of a report handler)
- Library
- PageHandler
- Program
- VGUIRecord

An EGL source file can include zero to many non-generatable parts but can include no more than one generatable part. The generatable part (if any) must be at the top level of the file and must have the same name as the file.

### Build file

An EGL build file (extension .eglbld) contains any number of build parts and is written in Extensible Markup Language (XML), in EGL build-file format. You can review the related DTD, which is in the following directory:

```
installationDir\egl\eclipse\plugins\
com.ibm.etools.egl_version
```

*installationDir*

The product installation directory, such as C:\Program Files\IBM\Rational\SPD\6.0. If you installed and kept a Rational® Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The installed version of the plugin; for example, 6.0.1

The file name (like egl\_wssd\_6\_0.dtd) begins with the letters *egl* and an underscore. The characters *wssd* refer to Rational Web Developer and Rational Application Developer; the characters *wsed* refer to Rational Application Developer for z/OS®; and the characters *wdsc* refer to Rational Application Developer for iSeries™.

After you add parts to files, you can use a repository to maintain a history of changes.

## Recommendations

This section gives recommendations for setting up your development projects.

### For build descriptors

Team projects should appoint one person as a build-descriptor developer. The tasks for that person are as follows:

- Create the build descriptors for the source-code developers
- Put those build descriptors in a project separate from the source code projects; and make that separate project available in the repository or by some other means
- Ask the source-code developers to set the property **default build descriptors** in their projects, so that the property references the appropriate build descriptors
- If a small subset of the build descriptor options (such as for user ID and password) varies from one source-code developer to the next, ask each source-code developer to do as follows:

- Code a personal build descriptor that uses the option **nextBuildDescriptor** to point to a group build descriptor
- Ask the source-code developers to set the property **default build descriptors** in their files, folders, or packages, so that the property references the personal build descriptor. They do not specify the property at the project level because the project-level property is under repository control, along with other project information.

For additional information, see *Build descriptor part*.

## For packages

For packages, recommendations are as follows:

- Do not use the same package name in different projects or source directories
- Do not use the default package

## Part assignment

For parts, many of the recommendations refer to good practices, not hard requirements. Fulfill even the optional recommendations unless you have good reason to do otherwise:

- A *requirement* when you use PageHandlers is that you put JSPs in the same project as their associated PageHandlers.
- If a non-generatable part (like a record part) is used only by one program, library, or PageHandler, place the non-generatable part in the same file as the using part.
- If a part is referenced from different files in the same package, put that part in a separate file in the package.
- If a part is shared across packages in a single project, place that part in a separate package in that project.
- Put code for completely unrelated applications in different projects. The project is the unit for transferring code between your local directory structure and the repository. Design project structure so that developers can minimize the amount of code they have to have loaded into their development system.
- Name projects, packages, and files in a way that reflects the use of the parts they contain.
- If your process emphasizes code ownership by a developer, do not assign parts for different owners to the same file.
- Assign parts to packages with a clear understanding of the purpose of the package; and group those parts by the closeness of the relationship between them.

The following distinction is important:

- Moving a part from file to file in the same package does not require that you change import statements in other files.
- Moving a part from one package to another may require an import statement to be added or changed in every file that references the moved part.

## Related concepts

“Build descriptor part” on page 383

“EGL services and Web services” on page 158

“Generation in the workbench” on page 416

“References to parts” on page 23

“Import” on page 33

“Introduction to EGL” on page 1

“Parts” on page 19

## Related reference

“EGL build-file format” on page 462

“EGL source format” on page 586

“EGL statements” on page 88

---

## Parts

An EGL file contains a set of *parts*, each of which is a discrete, named unit. Some parts (such as a program) are *generatable parts*; each of these is the basis of a compilable unit. A generatable part must have the same name as the EGL source file that contains the part; and in this case, the case of the name is significant: *myPart* is different from *MYPART*.

An EGL source file (extension .egl) can include zero or one generatable part and zero to many other parts.

Parts are also categorized in this way:

- *Logic parts* define a runtime sequence that you write in the EGL procedural language--
  - The non-generatable part *function* is the basic unit of logic. Every other kind of logic part can include functions.
  - You can define any of the following types of *programs*, which vary by interface type. Each is a generatable part:
    - A *basic program* either avoids interacting with the user or limits that interaction to a particular kind of character-based interface. The interface technology in this case works as follows:
      - Displays output in a command window; and
      - Allows the user to interact with the program in an immediate way, with each keystroke potentially defining a separate event for the program to handle.

For details on this kind of interface, see *Console user interface*.

- A *textUI program* interacts with the user in this way:
  - Displays a set of fields in a command window; and
  - Accepts the user’s field input only when the user presses a submit key.
- A *VGWebTransaction program* delivers interactive pages to Web browsers. The VGWebTransaction program is available primarily to support the migration of Web transactions from VisualAge Generator. As noted later, the EGL PageHandler is available to support development of new Web applications.

You can define any of the previous types of programs to be a *main program*. That kind of program is started by a program transfer other than a call, by an operating-system process, or in some cases by the user. In addition, you can declare a basic or textUI program to be a *called program*, which can be invoked only by a call.

For other details on the runtime deployment of main and called programs, see *runtime configurations*.

- A *PageHandler* is a generatable part that controls the interaction between the user and a Web page.
- A *Handler* of type JasperReport is a generatable part that contains customized functions which are invoked at different times during execution of a

JasperReports design file. The data returned from each function is included in your output report, which can be rendered in PDF, XML, text, CSV, or HTML format.

- A *Service* is a generatable part containing code that will be accessed as follows:
  - From EGL code by way of a TCP/IP connection (in which case the part contains the implementation details of an *EGL service*); or
  - From any code by way of an HTTP connection (in which case the part contains the implementation details of an *EGL Web service*).
- A *Library* is also a generatable part; a collection of shared functions and variables that can be made available locally to programs, pageHandlers, EGL services, and other libraries.
- *Data parts* define the data structures that are available to your program.

The following kinds of data parts are used as types in variable declarations:

- *DataItem parts* contain information about the most elementary kind of data. These parts are similar to entries in a system-wide data dictionary, with each part including details on data size, type, formatting rules, input-validation rules, and display suggestions. You define a *DataItem* part once and can use it as the basis for any number of primitive variables or record fields.

The *DataItem* part gives you a convenient way to create a variable from a primitive type. For example, consider the following definition of *myStringPart*, which is a *DataItem* part of type *String*:

```
DataItem
  MyStringPart String { validValues = ["abc", "xyz"] }
end
```

When you develop a function, you can declare a variable of type *MyStringPart*:

```
myString MyStringPart;
```

The following declaration has the same effect as the previous one:

```
myString STRING { validValues = ["abc", "xyz"] };
```

As shown, the name of a *DataItem* part is simply an alias for a primitive type that has specific property settings.

- *Record parts* are a basis for complex data. A variable whose type is a record part includes fields and is called a record.

Two categories of record parts are available: fixed and non-fixed. The latter category is more widely used, and we describe it first.

Each field in a non-fixed record part can be based on any of these:

- A primitive type such as *STRING*
- A *DataItem* part
- A fixed-record part (as described later)
- Another record part
- An array of any of the preceding kinds

Each field also can be a *Dictionary* or *ArrayDictionary* (as described later); or an array of *Dictionaries* or *ArrayDictionaries*.

You can use a non-fixed record part to create variables for general processing or to access a relational database.

The length of the data in the non-fixed record can vary at run time.

- *Fixed record parts* are a basis for complex data that is of fixed length. A variable whose type is a fixed record part includes fields, and each field can have any of these as a type:



- A primitive type such as CHAR
- A dataItem part

Each field can be substructured. For example, a field that specifies a telephone number can be defined as follows:

```
10 phoneNumber    CHAR(10);
20 areaCode       CHAR(3);
20 localNumber    CHAR(7);
```

Although you can use fixed record parts for any kind of processing, their best use is for I/O operations on VSAM files, MQSeries® messages queues, and other sequential files.

To some extent, EGL supports fixed record parts to allow compatibility with earlier products such as VisualAge Generator. Although you can use fixed records for accessing relational databases or for general processing, it is recommended that you avoid using fixed records for those purposes.

- A *Dictionary part* is always available; you do not define it. A variable that is based on a dictionary part may include a set of keys and their related values, and you can add and remove key-and-value entries at run time.
- An *ArrayDictionary part* is always available; you do not define it. A variable that is based on an ArrayDictionary part lets you access a series of arrays by retrieving the same-numbered element of every array. A set of elements that is retrieved in this way is itself a dictionary, with each array name treated as a key that is paired with the value in the array element.

An ArrayDictionary is especially useful in relation to the display technology described in *Console user interface*.

The other data part is a *DataTable*, which is treated as a variable rather than as a type for a variable. The DataTable is a generatable part that can be shared by multiple programs. It contains a series of rows and columns; includes a primitive value in each cell; and is treated as a variable that is (in most cases) global to the run unit.

- *UI (user interface) parts* describe the layout of data presented to the user in fixed-font screen and print forms. UI parts are used in different contexts and are of the following types:
  - A *record part* of subtype *ConsoleForm* is an organization of data that is presented to the user in the context of consoleUI technology. Like other record parts, each is used as a type for one or more variables; but in this case, each variable is called a *console form* rather than a record. The ConsoleUI technology also includes other parts that are defined for you and can be used as the basis of variables; for details, see *Console user interface*.
  - A *Form* is also an organization of data that is presented to the user. One kind of form organizes the data sent to a screen in a textUI program, and another organizes the data sent to a printer in any kind of program.

Each form includes a fixed, internal structure like that of a fixed record; but a form cannot include a substructure.

A form is made available to a Program, PageHandler, or Library only if the form is included or referenced by a FormGroup, as described next.

- A *FormGroup part* is a collection of text and print forms and is a generatable part. A program can include only one formGroup for most uses, along with one formGroup for help-related output. The same form can be included in multiple FormGroups.

The forms in a FormGroup are global to a program, though access must be specified in a program-specific use statement. The forms are referenced as variables.

- A *record part* of subtype *VGUIRecord* is an organization of data that is presented to the user in the context of a *VGWebTransaction* program.

You create Web user interfaces with Page Designer, which builds a JSP file and (when you are not working on a *VGWebTransaction* program) associates the JSP file with an EGL PageHandler. The JSP file is similar to a UI part for applications that interact with the user by way of the Web.

For details on Web access, see *Web support*.

- *Access parts* allow you to interact with external code:
  - An *Interface part* is a non-generatable part that allows you to access functionality from an EGL service, from a Web service (EGL or otherwise), or from Java code.
  - The *Report part* and *ReportData part* are predefined and are used as the basis of variables that help fill a report at run time. The report is based on an open-source, Java-based library called JasperReports. For an introduction to the EGL technology, see *EGL reports overview*.
- *Build parts* are defined in EGL build files (extension .eglbld) and define a variety of processing characteristics:
  - A *build descriptor part* controls the generation process and indicates what other control parts are read during that process.
  - A *linkage options part* gives details on how a generated program transfers to other programs. The information in this part is used at generation time, test time, and run time.
  - A *resource associations part* relates an EGL record with the information needed to access a file on a particular target platform; the information in this part is used at generation time, test time, and run time.

A fixed record, *DataTable*, or form (whether text or print) includes a *fixed structure*. The structure is composed of a series of fields, each of which has a size and type that is known at generation time; and in the case of a *DataTable* or fixed record, the field can be substructured.

## Related concepts

“*ArrayDictionary*” on page 87

“*Build descriptor part*” on page 383

“*Compatibility with VisualAge Generator*” on page 532

“*Console user interface*” on page 207

“*DataItem part*” on page 133

“*Dictionary*” on page 83

“*EGL projects, packages, and files*” on page 15

“*EGL reports overview*” on page 251

“*Fixed record parts*” on page 136

“*Function part*” on page 150

“*Import*” on page 33

“*Introduction to EGL*” on page 1

“*Linkage options part*” on page 399

“*Program part*” on page 148

“*Record parts*” on page 135

“*References to parts*” on page 23

“*References to variables in EGL*” on page 59

“*Resource associations and file types*” on page 393

“*Runtime configurations*” on page 11

“Fixed structure” on page 27  
“Typedef” on page 28  
“Web support” on page 215

#### **Related reference**

“EGL build-file format” on page 462  
“EGL editor” on page 577  
“EGL source format” on page 586  
“EGL statements” on page 88  
“Primitive types” on page 34

## **References to parts**

This section describes a set of rules that determine how EGL identifies the part to which a name refers. These rules are important in the following situations:

- One function invokes another
- A non-function part (a dataItem part, for example) refers to a validator function
- A part acts as a typedef (a model of format) in the declaration of a structure item or variable
- One part references another in a use declaration
- One build part references another

A second set of rules determines how EGL resolves variable references. For details, see *References to variables and constants*.

### **Basic visibility rules**

In the simplest case, you define parts one after the next in a single package, without declaring one part within another. The following list omits many details, but shows a series of parts that are at the same hierarchical level:

```
Function: Function01
Function: Function02
Function: Function03
Record:   Record01
```

Parts at the same level are available to one another. Function01, for example, can invoke one or both of the other functions; and Record01 can be used as a typedef for variables in each of the three functions.

In most cases, a part *cannot* nest another part. The exceptions are as follows:

- A program, library, or pageHandler can nest functions, but even then, the inclusion must be direct; a function cannot nest another function
- A form group can nest forms

An example with nested parts is as follows:

```
Program: Program01
  Function: Function01
  Function: Function02
Function: Function03
Record:   Record01
```

Parts at the top level are available to every other part in the package. However, the nested parts (Function01 and Function02) are available only to a subset of parts in the package:

- They are available to each other.

- They are available to the nesting part and to functions that are used by the nesting part at run time. If Function01 invokes Function03, for example, Function03 can invoke Function02 because Function03 is used in Program01.

Finally, if your code includes text or print forms, a use declaration is necessary to access the form group that includes those forms. A use declaration is also desirable when accessing data tables or libraries. For additional information, see *Use declaration*.

## Additional visibility rules

Most development efforts have parts that are shared across more than one package. These rules are in effect:

- Any part in the file can reference parts from other packages, so long as the accessed parts have these characteristics:
  - Are top-level parts
  - Are not declared as *private*
  - Are either in the same project as the referencing part or are in a project listed in the EGL build path of the referencing project

You can provide access in these ways:

- You can qualify the part name with the package name, in which case no import statement is needed in your source file. If a package name is *my.package*, for example, and a part name is *myPart*, you can reference the part as follows:

```
my.package.myPart
```

- You can use import statements, which provide the following benefits:
  - Import statements make it possible for you to avoid qualifying the names of the imported parts, unless you must use a package name to avoid an ambiguous reference.
  - Import statements provide a way to document what packages are used in the source code.

## Part-name resolution

To resolve a part reference, EGL conducts a search that includes one to many steps. The following statements apply *at each step*:

- The search ends successfully if a uniquely named part is found
- The search ends with an error if two same-named parts are found.

These situations are possible:

- The part reference is qualified with a package name; in this case, the search always includes only one step
- The part reference is not qualified with a package name and is not a function invocation
- The part reference is not qualified with a package name and is a function invocation

The next statements are rarely important, but could apply in either of the last two situations:

- The property **containerContextDependent** in the referencing function may be set to *yes*. You set that property to extend the name space used to resolve references, as described in *containerContextDependent*.

- If one of your functions is visible to the program or PageHandler and has a name that is identical to the name of an EGL system function, your function is referenced rather than the system function.

**Part-name resolution when the package name is specified:** As noted earlier, you can specify the name of a package when referencing a part, as in the example `my.package.myPart`. The current project is considered, as are any projects listed in the EGL build path.

If the reference is from a part that is inside the same package, the following statements apply:

- The package name is valid but unnecessary
- The part name is resolved even if the part is declared as private

**Part-name resolution (other than function invocation) when the package name is not specified:** If a part references a part other than a function and does not specify a package name, the steps in the search order are as follows:

1. Search the parts nested in the same container as the one in which the referencing part is nested.
2. Search the parts that were explicitly imported in the file where the referencing part resides. The current project is considered, as are any projects listed in the EGL build path.

Each import statement in this case explicitly references a particular part in a particular package. The part named in such an *explicit-type import statement* acts as an override of the same-named part in the current package.

If you have identically named packages in two different projects, a given explicit-type import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to a package in a given project.) The presence of a same-named package in two different projects is not an error, but creates a confusing situation and is not recommended.

An error occurs if you have two explicit-type import statements that name the same part.

3. Search the top-level parts that are in the same package as the referencing part. The current project is considered, as are any projects listed in the EGL build path. Finding two parts of the same name causes an error.
4. Search other imported parts. The current project is considered, as are any projects listed in the EGL build path.

Each import statement in this case references all parts in a given package and is called a *wild-card import statement*.

If you have identically named packages in two different projects, a given wild-card import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to a package in a given project.)

If more than one wild-card import statement retrieves the same-named part, an error occurs.

**Function invocation when the package name is not specified:** If a part invokes a function and does not specify a package name, the steps in the search order are as follows:

1. Search the functions nested in the same container as the one in which the invoker is nested.

2. Search the functions residing in the libraries specified in the container's use declarations.
3. Continue the search only with functions that are being included in the container at generation time. (To include functions other than those nested in the same container or residing in a library, set the container property **includeReferencedFunctions** to *yes*.)

The search of the included functions occurs as follows:

- a. Search the parts that were explicitly imported in the file where the container resides. The current project is considered, as are any projects listed in the EGL build path.

Each import statement in this case explicitly references a particular part in a particular package. The part named in such an *explicit-type import statement* acts as an override of the same-named part in the current package.

If you have identically named packages in two different projects, a given explicit-type import statement uses the EGL build path to do a first-found search, stopping when the required function is found. (The function must be unique to a package in a given project.) The presence of a same-named package in two different projects is not an error, but creates a confusing situation and is not recommended.

An error occurs if you have two explicit-type import statements that name the same part.

- b. Search the top-level functions in the same package as the container. The current project is considered, as are any projects listed in the EGL build path. An error occurs if the search finds two parts of the same name.
- c. Search other imported parts. The current project is considered, as are any projects listed in the EGL build path.

Each import statement in this case references all parts in a given package and is called a *wild-card import statement*.

If you have identically named packages in two different projects, a given wild-card import statement uses the EGL build path to do a first-found search, stopping when the required part is found. (The part must be unique to a package in a given project.)

If more than one wild-card import statement retrieves the same-named part, an error occurs.

## Program invocation

When a program is invoked on a **call** or **transfer** statement, the argument list of the invoker must match the parameter list of the invoked program. A mismatch of argument and parameter causes an error.

### Related concepts

"EGL projects, packages, and files" on page 15

"Import" on page 33

"Introduction to EGL" on page 1

"Parts" on page 19

"References to variables in EGL" on page 59

### Related reference

"containerContextDependent" on page 557

"EGL build path and eglpath" on page 571

"EGL editor" on page 577

"EGL source format" on page 586

"Use declaration" on page 1091

## Fixed structure

A *fixed structure* establishes the format of a text form, print form, dataTable, or fixed-record part; and is composed of a series of fields that each describes an elemental memory location or a collection of memory locations, as in this example:

```
10 workAddress;  
  20 streetAddress1 CHAR(20);  
    30 Line1 CHAR(10);  
    30 Line2 CHAR(10);  
  20 streetAddress2 CHAR(20);  
    30 Line1 CHAR(10);  
    30 Line2 CHAR(10);  
  20 city CHAR(20);
```

You can define all the fields directly in the definition, as in the preceding example. Alternatively, you can indicate that all or a subset of the structure is equivalent to the structure that is in another fixed record part; for details, see *Typedef*.

Access to a field is based on a variable name, then a series of field names with a dot syntax. If you declare that the record *myRecord* includes the structure shown in the previous example, each of the following identifiers refers to an area of memory:

```
myRecord.workAddress  
myRecord.workAddress.streetAddress1  
myRecord.workAddress.streetAddress1.Line1
```

An *elementary structure field* has no subordinate structure fields and describes an area of memory in either of these ways:

- By a specification of length and primitive type, as in the previous example; or
- By pointing to the declaration of a dataItem part, as described in *Typedef*.

As shown earlier, a field in a fixed structure can have subordinate fields. Consider the next example:

```
10 topMost;  
  20 next01 HEX(4);  
  20 next02 HEX(4);
```

When you define a superior structure field (like *topMost*), you have several options:

- If you do not assign a length or primitive type, the superior structure field is of type CHAR, and EGL calculates the length. The primitive type of *topMost* is CHAR, for example, and the length is 4.
- If you assign a primitive type but do not assign a length, EGL calculates the length based on characteristics of the subordinate structure items
- If you assign both a length and primitive type, the length must reflect the space provided for the subordinate structure fields; otherwise, an error occurs

**Note:** The primitive type of a fixed-structure field determines the number of bytes in each unit of length; for details, see *Primitive types*.

Each elementary structure field has a series of properties, whether by default or as specified in the structure field. (The structure field may refer to a dataItem part that itself has properties.) For details, see *Overview of EGL properties and overrides*.

### Related concepts

“DataItem part” on page 133  
“Fixed record parts” on page 136  
“Overview of EGL properties” on page 64



“Parts” on page 19  
“References to variables in EGL” on page 59  
“Typedef”

#### Related reference

“Data initialization” on page 564  
“EGL source format” on page 586  
“Primitive types” on page 34  
“SQL item properties” on page 68

## Typedef

A type definition (typedef) is a part that is used as a model of format. You use the typedef mechanism for these reasons:

- To identify the characteristics of a variable
- To reuse part declarations
- To enforce formatting conventions
- To clarify the meaning of data

Often, typedefs identify an abstract grouping. You can declare a record part named *address*, for example, and divide the information into *streetAddress1*, *streetAddress2*, and *city*. If a personnel record includes the structure items *workAddress* and *homeAddress*, each of those structure items can point to the format of the record part named *address*. This use of typedef ensures that the address formats are the same.

Within the set of rules described in this page, you may point to the format of a part either when you declare another part or when you declare a variable.

When you declare a part, you are not required to use a part as a typedef, but you may want to do so, as in the examples that are shown later. Also, you are not required to use a typedef when you declare a variable that has the characteristics of a data item; instead, you can specify all characteristics of the variable, without reference to a part.

A typedef is *always* in effect when you declare a variable that is more complex than a data item. For instance, if you declare a variable named **myRecord** and point to the format of a part named **myRecordPart**, EGL models the declared variable on that part. If you point instead to the format of a part named **myRecordPart02**, the variable is called **myRecord** but has all characteristics of the part named **myRecordPart02**.

The table and sections that follow give details on typedefs in different contexts.

Entry that points to a typedef	Type of part to which the typedef can refer
function parameter or other function variable	a record part or dataItem part
program parameter	dataItem part, form part, record part
program variable (non-parameter)	dataItem part, record part
structure item	dataItem part, record part



## DataItem part as a typedef

You can use a dataItem part as a typedef in the following situations:

- When declaring a variable or parameter
- When declaring a structure item, which is a subunit of a record part, form part, or dataTable part

These rules apply:

- If a structure item is a parent to other structure items that are listed in the same declaration, the structure item can point only to the format of a dataItem part, as in this example:

```
DataItem myPart CHAR(20) end

Record myRecordPart type basicRecord
  10 mySI myPart; // myPart acts as a typedef
  20 a CHAR(10);
  20 b CHAR(10);
end
```

The previous record part is equivalent to this declaration:

```
Record myRecordPart type basicRecord
  10 mySI CHAR(20);
  20 a CHAR(10);
  20 b CHAR(10);
end
```

- You cannot use a dataItem part as a typedef and also specify the length or primitive type of the entity that is pointing to the typedef, as in this example:

```
DataItem myPart HEX(20) end

// NOT valid because mySI has a primitive type
// and points to the format of a part (to myPart, in this case)
Record myRecordPart type basicRecord
  10 mySI CHAR(20) myPart;
end
```

- A variable declaration that does not refer to a record part either points to the format of a dataItem part or has primitive characteristics. (A program parameter can refer to a form part, too.) A dataItem part, however, cannot point to the format of another dataItem part or to any other part.
- An SQL record part can use only the following types of parts as typedefs:
  - Another SQL record part
  - A dataItem part

## Record part as a typedef

You can use a record part as a typedef in the following situations:

- When declaring a structure item
- When declaring a variable (including a parameter), in which case the variable reflects the typedef in these ways:
  - Format
  - Record type (for example, indexedRecord or serialRecord)
  - Property values (for example, value of the **file** property)

When you declare a structure item that points to the format of another part, you specify whether the typedef adds a level of hierarchy, as illustrated later.

These rules apply:

- A record part can be a typedef when you use a structure item to facilitate reuse--

```

Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

```

```

Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 homeAddress address;
end

```

The second record part is equivalent to this declaration--

```

Record record1 type serialRecord
{ fileName = "myFile" }
  10 person CHAR(30);
  10 homeAddress;
    20 streetAddress1 CHAR(30);
    20 streetAddress2 CHAR(30);
    20 city CHAR(20);
end

```

If a structure item uses the previous syntax to point to the format of a structure part, EGL adds a hierarchical level to the structure part that includes the structure item. For this reason, the internal structure in the previous example has a structure-item hierarchy, with *person* at a different level from *streetAddress1*.

- In some cases, you prefer a flat arrangement in the structure; and an SQL record that is an I/O object for relational-database access *must* have such an arrangement--
  - In the previous example, if you substitute the word **embed** for a record part's structure item name (in this case, *homeAddress*) and follow that word with the name of the record part that acts as a typedef (in this case, *address*), the part declarations look like this:

```

Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

```

```

Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 embed address;
end

```

The internal structure of the record part is now flat:

```

Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

```

The only reason to use the word **embed** in place of a structure item name is to avoid adding a level of hierarchy. A structure item identified by the word **embed** has these restrictions:

- Can point to the format of a record part, but not to a dataItem part

- Cannot specify an array or include a primitive-type specification
- Next, consider the case in which a record part is a typedef when you are declaring identical structures in two records--

```
Record common type serialRecord
{
  fileName = "mySerialFile"
}
10 a BIN(10);
10 b CHAR(10);
end

Record recordA type indexedRecord
{
  fileName = "myFile",
  keyItem = "a"
}
  embed common; // accepts the structure of common,
                // not the properties
end

Record recordB type relativeRecord
{
  fileName = "myOtherFile",
  keyItem = "a"
}
  embed common;
end
```

The last two record parts are equivalent to these declarations--

```
Record recordA type indexedRecord
{
  fileName = "myFile",
  keyItem = "a"
}
10 a BIN(10);
10 b CHAR(10);
end

Record recordB type relativeRecord
{
  fileName = "myOtherFile",
  keyItem = "a"
}
10 a BIN(10);
10 b CHAR(10);
end
```

- You can use a record part multiple times as a typedef when declaring a series of structure items. This reuse makes sense, for example, if you are declaring a personnel record part that includes a home address and a work address. A basic record could provide the same format in two locations in the structure:

```
Record address type basicRecord
10 streetAddress1 CHAR(30);
10 streetAddress2 CHAR(30);
10 city CHAR(20);
end

Record record1 type serialRecord
{
  fileName = "myFile"
}
10 person CHAR(30);
10 homeAddress address;
10 workAddress address;
end
```

The record part is equivalent to this declaration:

```
Record record1 type serialRecord
{
  fileName = "myFile"
}
10 person CHAR(30);
10 homeAddress;
20 streetAddress1 CHAR(30);
20 streetAddress2 CHAR(30);
20 city CHAR(20);
10 workAddress;
20 streetAddress1 CHAR(30);
20 streetAddress2 CHAR(30);
20 city CHAR(20);
end
```

- You cannot use a record part as a typedef and also specify the length or primitive type of the entity that is pointing to the typedef, as in this example:

```
Record myTypedef type basicRecord
10 next01 HEX(20);
10 next02 HEX(20);
end

// not valid because myFirst has a
// primitive type and points to the format of a part
Record myStruct02 type serialRecord
{
  fileName = "myFile"
}
10 myFirst HEX(40) myTypedef;
end
```

Consider the following case, however:

```
Record myTypedef type basicRecord
10 next01 HEX(20);
10 next02 HEX(20);
end

Record myStruct02 type basicRecord
10 myFirst myTypedef;
end
```

The second structure is equivalent to this declaration:

```
Record myStruct02 type basicRecord
10 myFirst;
20 next01 HEX(20);
20 next02 HEX(20);
end
```

The primitive type of any structure item that has subordinate structure items is CHAR by default, and the length of that structure item is the number of bytes represented by the subordinate structure items, regardless of the primitive types of those structure items. For other details, see *Structure*.

- The following restrictions are in effect in relation to SQL records:
  - If an SQL record part uses another SQL record part as a typedef, each item provided by the typedef includes a four-byte prefix. If a non-SQL record uses an SQL record part as a typedef, however, no prefix is included. For background information, see *SQL record internals*.
  - An SQL record part can use only the following types of parts as typedefs:
    - Another SQL record part
    - A dataItem part
- Finally, neither a structure nor a structure item can be a typedef

## Form as a typedef

You can use a form part as a typedef only when declaring a program parameter.

### Related concepts

“DataItem part” on page 133  
“Form part” on page 184  
“Introduction to EGL” on page 1  
“Record parts” on page 135  
“Fixed structure” on page 27

### Related tasks

“Creating an EGL program part” on page 147

### Related reference

“EGL statements” on page 88  
“SQL record internals” on page 876

---

## Import

An import statement identifies a set of parts that are in a specified package (for EGL source files) or in a specified set of files (for EGL build files). The file that holds an import statement can reference the imported parts as if they were in the same package as the file.

## Background

If a public part resides in a package other than the current one but is not identified in an import statement, your code needs to qualify the part name (for example, myPart) with the package name (for example, my.pkg), as in this example:

```
my.pkg.myPart
```

If the part is identified in an import statement, however, your code can drop the package name. In this case, the unqualified part name (like myPart) is sufficient.

For a description of the circumstances in which import statements are used to resolve a part name, see *References to Parts*.

## Format of the import statement

The syntax is as follows for the import statement in an EGL source file:

```
import packageName.partSelection;
```

*packageName*

Identifies the name of a package in which to search. The name must be complete.

*partSelection*

Is a part name or an asterisk (\*). The asterisk indicates that all parts in the package are selected.

An import statement in a build file identifies other build files whose parts can be referenced by parts in the importing file. The import statements follow the <EGL> tag in the build file, and each statement has the following syntax:

```
<import file=filePath.eglbld>
```

*filePath*

Identifies the path and name of the file to import. If you specify a path, the following statements apply:

- The file path is in any of the source directories in the same project or in any other project that is in the EGL path
- Each qualifier is separated from the next by a virgule (/)

You may specify an asterisk (\*) as the file name or as the last character of the file name. If the asterisk is used, EGL imports all the .eglbld files with these characteristics:

- Are in the specified file path.
- Have names that begin with the characters that precede the asterisk. (If the asterisk has no preceding characters, all build files in the directory path are selected.)

The file extension .eglbld is optional.

#### **Related concepts**

“EGL projects, packages, and files” on page 15

“Introduction to EGL” on page 1

“Parts” on page 19

“References to parts” on page 23

#### **Related tasks**

“Editing an EGL build path” on page 407

---

## **Primitive types**

Each EGL primitive type characterizes an area of memory. There are three kinds of primitive types: character, numeric, and datetime.

- The character types are as follows:
  - *CHAR* refers to single-byte characters.
  - *DBCHAR* refers to double-byte characters. *dbchar* replaces *DBCS*, which was a primitive type in EGL V5.
  - *MBCHAR* refers to multibyte characters, which are a combination of single-byte and double-byte characters. *mbchar* replaces *MIX*, which was a primitive type in EGL V5.
  - *STRING* refers to a field of varying length, where the double-byte characters conform to the UTF-16 encoding standards developed by the Unicode Consortium.
  - *UNICODE* refers to a fixed field, where the double-byte characters conform to the UTF-16 encoding standards developed by the Unicode Consortium.
  - *HEX* refers to hexadecimal characters.
- The datetime types are as follows:
  - *DATE* refers to a specific calendar date that has a fixed length of eight single-byte digits.
  - *INTERVAL* refers to a span of time that has a length ranging from two to twenty-seven single-byte digits.
  - *TIME* refers to an instance in time that has a fixed length of six single-byte digits.
  - *TIMESTAMP* refers to the current time and has a length ranging from two to twenty single-byte digits.
- The large object types are as follows:
  - *BLOB* refers to a binary large object with a length ranging from one byte to two gigabytes.

- *CLOB* refers to a character large object with a length ranging from one byte to two gigabytes.
- The numeric types are as follows:
  - *BIGINT* refers to an 8-byte area that stores an integer of as many as 18 digits. This type is equivalent to type *BIN*, length 8, no decimal places.
  - *BIN* refers to a binary number.
  - *DECIMAL* refers to packed decimal characters whose sign is represented by a hexadecimal C (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte. *DECIMAL* replaces *PACK*, which was a primitive type in EGL version 5.0.
  - *FLOAT* refers to an 8-byte area that stores a double-precision floating-point numbers with up to 16 significant digits.
  - *INT* refers to a 4-byte area that stores an integer of as many as 9 digits. This type is equivalent to type *BIN*, length 4, no decimal places.
  - *MONEY* refers to currency amounts, which are stored as *DECIMAL* values.
  - *NUM* refers to numeric characters whose sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. For ASCII, that value is 3 (for a positive number) and 7 (for negative); for EBCDIC, that value is F (for a positive number) and D (for negative).
  - *NUMC* refers to numeric characters whose sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. For ASCII, that value is 3 (for a positive number) and 7 (for negative); for EBCDIC, that value is F (for a positive number) and C (for negative).
  - *PACF* refers to packed decimal characters whose sign is represented by a hexadecimal F (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte.
  - *SMALLFLOAT* refers to a 4-byte area that stores a single-precision floating-point number with up to 8 significant digits.
  - *SMALLINT* refers to an 2-byte area that stores an integer of as many as 4 digits. This type is equivalent to type *BIN*, length 2, no decimal places.

The internal representation of a field of any of the fixed-point numeric types is the same as an integer representation, even when you specify a decimal point. The representation of 12.34 is the same as that of 1234, for example. Similarly, currency symbols are not stored with fields of type *MONEY*.

When you interact with DB2® (directly or by way of JDBC) , the maximum number of digits in a fixed-point number is 31 at most.

A variable of type *ANY* receives the type of the value that is assigned to that variable, as described in the topic *ANY*.

At declaration time, you specify the primitive type that characterizes each of these values:

- The value returned by a function
- The value in a field, which is an area of memory that is referenced by name and contains a single value

Other entities also have a primitive type:

- A system variable has a primitive type (usually *NUM*) that is specific to the field
- A character literal is of one of these types:
  - *CHAR* if the literal includes only single-byte characters



- DBCHAR if the literal includes only double-byte characters from the double-byte character set
- MBCHAR if the literal includes a combination of single-byte and double-byte characters
- Character literals of type UNICODE are not supported.

Each primitive type is described on a separate page; and additional details are available on the pages that cover assignments, logical expressions, function invocations, and the call statement.

The sections that follow cover these subjects:

- Primitive types at declaration time
- Relative efficiency of different numeric types

## Primitive types at declaration time

Consider the following declarations:

```
DataItem
  myItem CHAR(4)
end
Record mySerialRecordPart type serialRecord
{
  fileName="myFile"
}
10 name CHAR(20);
10 address;
20 street01 CHAR(20);
20 street02 CHAR(20);
end
```

As shown, you must specify a primitive type when you declare these entities:

- A primitive variable
- A structure field that is not substructured

You may specify the primitive type of a substructured structure field like *address*. If you fail to specify the primitive type of such a structure field but you reference the structure field in your code, the product makes these assumptions:

- The primitive type is assumed to be CHAR, even if the subordinate structure fields are of a different type
- The length is assumed to be the number of bytes in the subordinate structure fields

## Relative efficiency of different numeric types

EGL supports the types DECIMAL, NUM, NUMC, and PACF so you can work more easily with files and databases that are used by legacy applications. It is recommended that you use fields of type BIN in new development or that you use an equivalent integer type (BIGINT, INT, or SMALLINT); calculations are most efficient with fields of those types. You get the greatest efficiency by using fields of type BIN, length 2, and no decimal places (the equivalent of type SMALLINT).

In calculations, assignments, and comparisons, fields that are of type NUM and have no decimal places are more efficient than fields that are of type NUM and have decimal places.

Calculations with fields of types DECIMAL, NUM, NUMC, and PACF are equally efficient.

### Related concepts

"DataItem part" on page 133  
"Record parts" on page 135  
"References to variables in EGL" on page 59  
"Fixed structure" on page 27

### Related reference

"ANY"  
"Assignments" on page 456  
"BIN and the integer types" on page 50  
"call" on page 665  
"CHAR" on page 38  
"DATE" on page 41  
"DBCHAR" on page 38  
"DECIMAL" on page 50  
"Exception handling" on page 94  
"FLOAT" on page 51  
"Function invocations" on page 613  
"HEX" on page 38  
"INTERVAL" on page 42  
"Logical expressions" on page 593  
"MBCHAR" on page 39  
"MONEY" on page 51  
"NUM" on page 51  
"NUMC" on page 52  
"Numeric expressions" on page 600  
"Operators and precedence" on page 779  
"PACF" on page 52  
"SMALLFLOAT" on page 53  
"SQL item properties" on page 68  
"STRING" on page 40  
"Text expressions" on page 601  
"TIME" on page 44  
"TIMESTAMP" on page 44  
"UNICODE" on page 41

## ANY

A variable of type ANY receives the type of the value that is assigned to that variable. The value can be of a primitive type such as INT or can be a variable that is based on a data part used as a type. The value cannot be a form or dataTable.

Consider this example:

```
myInt INT = 1;
myString STRING = "EGL";

myAny01, myAny02 any;

// myAny01 receives the value 1 and the type INT
myAny01 = myInt;

// myAny02 receives the value "EGL" and the type STRING
myAny02 = myString;

// The next statement is
// NOT VALID because a variable of type INT
// is being assigned to a variable of type STRING
myAny02 = myAny01;
```

Actions that combine types in an invalid way are detected only at run time and cause program termination. Those actions include assigning a value to a field of an incompatible type, passing an argument value to a parameter of an incompatible type, or combining incompatible values inside an expression.

The type of a literal is implied by the value of that literal:

- A quoted string is of type `STRING`
- An integer of 4 digits or less is of type `SMALLINT`
- An integer of 5 to 8 digits is of type `INT`
- An integer of 9 to 18 digits is of type `BIGINT`
- A number that includes a decimal point is of type `NUM`

When you reference a variable of type `ANY`, access is always dynamic. You cannot include a field of type `ANY` in a fixed structure (a `dataTable`, print form, text form, or fixed record).

**Related reference**

“Primitive types” on page 34

## Character types

### **CHAR**

An item of type `CHAR` is interpreted as a series of single-byte characters. The length reflects both the number of characters and the number of bytes and ranges from 1 to 32767.

Workstation platforms like Windows 2000 use the ASCII character set; mainframe platforms like z/OS UNIX<sup>®</sup> System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

**Related reference**

“Primitive types” on page 34

### **DBCHAR**

An item of type `DBCHAR` is interpreted as a series of double-byte characters. The length reflects the number of characters and ranges from 1 to 16383. To determine the number of bytes, double the length value.

Workstation platforms like Windows 2000 use the ASCII character set; mainframe platforms like z/OS UNIX System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

DBCS data is ideographic, as is necessary to display Chinese, Japanese, or Korean, for example. Display of such data requires a terminal device with DBCS capability.

**Related reference**

“Primitive types” on page 34

### **HEX**

An item of type `HEX` is interpreted as a series of hexadecimal digits (0-9, a-f, and A-F), which are treated as characters. The length reflects the number of digits and ranges from 1 to 65534. To determine the number of bytes, divide by 2.

For an item of length 4, the internal bit representations of example values are as follows:

```
// hexadecimal value 04 D2
00000100 11010010

// hexadecimal value FB 2E
11111011 00101110
```

The primary use of an item of type HEX is to access a file or database field whose data type does not match another EGL primitive type.

You can assign a hexadecimal value by using a literal that is of type CHAR and that includes only characters in the range of hexadecimal digits, as in these examples:

```
myHex01 = "ab02";

myHex02 = "123E";
```

You can include a hexadecimal item as an operand in a logical expression, as in these examples:

```
if (myHex01 = "aBCd")
  myFunction01();
else
  if (myHex > myHex02)
    myFunction02();
  end
end
```

You cannot include a hexadecimal item in an arithmetic expression.

#### **Related reference**

“Primitive types” on page 34

### **MBCHAR**

An item of type MBCHAR is interpreted as a combination of single-byte and double-byte characters. The length reflects the number of single-byte characters that the item can contain and also reflects the number of bytes. The length ranges from 1 to 32767.

Workstation platforms like Windows 2000 use the ASCII character set; mainframe platforms like z/OS UNIX System Services use the EBCDIC character set. Differences in collating sequence generally cause greater-than and less-than comparisons to have different results in the two types of environments.

On a mainframe environment, you must include space for shift-out and shift-in characters if double-byte characters are possible in the item:

- A single-byte shift-out character (hex value 0E) indicates the beginning of a series of double-byte characters
- A single-byte shift-in character (hex value 0F) indicates the end of that series

The shift-out and shift-in characters are deleted during an EBCDIC-to-ASCII data conversion and are inserted during an ASCII-to-EBCDIC data conversion. If a variable-length record is being converted, and if the current record end (as indicated by the record length) is within a structure item that is of type MBCHAR, the record length is adjusted to reflect the insertion or deletion of the shift-out and shift-in characters.

Double-byte character data is ideographic, as is necessary to display Chinese, Japanese, or Korean, for example. Display of such data requires a terminal device with double-byte character set capability.

#### Related reference

“Primitive types” on page 34

## STRING

The primitive type STRING is composed of double-byte UNICODE characters.

You can store the value of the field in a file or database. If your code interacts with DB2 UDB, you must ensure that the code page for GRAPHIC data is UNICODE and that the column that stores the data item value is of SQL data type GRAPHIC or VARGRAPHIC.

For details on Unicode, see the web site of the Unicode Consortium ([www.unicode.org](http://www.unicode.org)).

The following syntax declares a limited-length string, which is a string that is restricted to a specified number of characters:

```
varName STRING(lengthLimit);
```

*varName*

Name of the variable. For details on validity, see *Naming conventions*.

*lengthLimit*

An integer that represents the number of characters. The value is greater than zero.

When you specify a limited-length string as a function parameter whose modifier is OUT or INOUT, the length limit must be the same in argument and parameter. When you specify a limited-length string as a function parameter whose modifier is IN, any text input is valid. The following statements apply to the latter case as well as when you assign a value to a limited-length string:

- If more characters are in the source than are valid in the target, EGL runtime truncates the copied content to fit the available length.
- If fewer characters are in the source than are valid in the target, EGL runtime pads the copied content with blanks, to the specified length.

The following statements apply to comparisons:

- Trailing blanks in fields that are of type STRING (but are not of limited length) are considered during a comparison.
- Trailing blanks in a limited-length string are ignored during a comparison. Before the comparison, the other operand (if shorter than the limited-length field) is padded with blanks or binary zeros up to the number of characters declared in the field of type limited length.

When you concatenate a limited-length string with another string, the contribution of the limited-length string depends on the value of build descriptor option **itemsNullable**. If the value of that option is YES, the limited-length string is padded with blanks to the last position specified in the string declaration. Otherwise, no padding occurs.

For details on comparisons, see *Logical expressions*. Also see *Substrings*.

### Related reference

"Assignment compatibility in EGL" on page 451

"Logical expressions" on page 593

"Naming conventions" on page 778

"Primitive types" on page 34

"Substrings" on page 882

## UNICODE

The primitive type UNICODE gives you a way to process and store text that may be in any of several human languages; however, the text must have been provided from outside your code. Literals of type UNICODE are not supported.

The following statements are true of an item of type UNICODE:

- The length reflects the number of characters and ranges from 1 to 16383. The number of bytes reserved for such an item is twice the value you specify for length.
- The item can be assigned or compared only to another item of type UNICODE.
- All comparisons compare the bit values in accordance with the order of characters in the UTF-16 encoding standard.
- When necessary, EGL pads the item with Unicode blanks.
- The system string functions treat the item as a string of individual bytes, which include the added Unicode blanks, if any. Any lengths you specify in those functions must be in terms of bytes rather than in terms of characters.
- You can store the value of the item in a file or database. If your code interacts with DB2 UDB, you must ensure that the code page for GRAPHIC data is UNICODE and that the column that stores the data item value is of SQL data type GRAPHIC or VARGRAPHIC.

For details on Unicode, see the web site of the Unicode Consortium ([www.unicode.org](http://www.unicode.org)).

### Related reference

"Primitive types" on page 34

## DateTime types

### DATE

An item of type DATE is a series of eight single-byte numeric digits that reflect a specific calendar date.

The format of type DATE is *yyyyMMdd*:

*yyyy*

Four digits that represent a year. The range is 0000 to 9999.

*MM*

Two digits that represent a month. The range is 01 to 12.

*dd* Two digits that represent a day. The range is 01 to 31, and an error occurs if your code assigns an invalid date such as 20050230.

The internal hexadecimal representation of an example value is as follows if the item is on a host environment which uses EBCDIC:

```
// March 15, 2005  
F2 F0 F0 F5 F0 F3 F1 F5
```

The internal hexadecimal representation of an example value is as follows if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// March 15, 2005  
32 30 30 35 30 33 31 35
```

An item of type DATE can receive data from and provide data to a relational database.

#### **Related reference**

"EGL library DateTimeLib" on page 919

"Datetime expressions" on page 591

"Primitive types" on page 34

"Date, time, and timestamp format specifiers" on page 46

## **INTERVAL**

An item of type INTERVAL is a series of one to twenty-one single-byte numeric digits that reflect an interval, which is the numeric difference between two points in time. The meaning of each digit is determined by the mask that you specify when declaring the item.

An interval can be positive (as when 1980 is subtracted from 2005) or negative (as when 2005 is subtracted from 1980), and at the beginning of the item is an extra byte that is not reflected in the mask. If an item of type INTERVAL is in a record, you must account for that extra byte when calculating the length of the record as well as the length of the superior item, if any.

You can specify a mask that is in either of two formats:

- Month span, which can include years and months
- Second span, which can include days, hours, minutes, seconds, and fractions of seconds

In either case, each character in the mask represents a digit. In the month-span format, for example, the set of *y*'s indicate how many years are in the item. If you only need three digits to represent the number of years, specify *yyy* in the mask. If you need the maximum number of digits (nine) to represent the number of years, specify *yyyyyyyyyy*.

In a given mask, the first character may be used as many as nine times (unless otherwise stated); but the number of each subsequent kind of character is restricted further.

For a mask that is in month-span format, the following characters are available, in order:

*y* Zero to nine digits that represent the number of years in the interval.

*M* Zero to nine digits that represent the number of months in the interval. If *M* is not the first character in the mask, only two digits are allowed, at most.

The default mask is *yyyyMM*.

For a mask that is in second-span format, the following characters are available, in order:

*d* Zero to nine digits that represent the number of days in the interval.



- H* Zero to nine digits that represent the number of hours in the interval. If *H* is not the first character in the mask, only two digits are allowed, at most.
- m* Zero to nine digits that represent the number of minutes in the interval. If *m* is not the first character in the mask, only two digits are allowed, at most.
- s* Zero to nine digits that represent the number of seconds in the interval. If *s* is not the first character in the mask, only two digits are allowed, at most.
- f* Zero to six digits that each represent a fraction of seconds; the first represents tenths, the second represents hundreds, and so on. Even when *f* is the first character in the mask, only six digits are allowed, at most.

Although you can have zero characters of a given kind at the beginning or end of a mask, you cannot skip intermediate characters. Valid masks include these:

```
yyyyyyMM
yyyyyy
MM

ddHHmmssffffff
HHmmssff
mmss
HHmm
```

The following masks, however, are invalid because intermediate characters are missing:

```
// NOT valid
ddmmssffffff
HHssff
```

The internal hexadecimal representation of an example value is as follows if the default mask (*yyyyMM*) is in effect and if the item is on a host environment which uses EBCDIC:

```
// 100 years, 2 months; the 4E means the value is positive
4E F0 F1 F0 F0 F0 F2

// 100 years, 2 months; the 60 means the value is negative
60 F0 F1 F0 F0 F0 F2
```

The internal hexadecimal representation of an example value is as follows if the default mask (*yyyyMM*) is in effect and if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// 100 years, 2 months; the 2B means the value is positive
2B 30 31 30 30 30 32

// 100 years, 2 months; the 2D means the value is negative
2D 30 31 30 30 30 32
```

An item of type INTERVAL is strongly typed, so you cannot compare an item of this type with an item of any other type; nor can you assign an item of any other type to or from an item of this type.

Finally, an item of type INTERVAL cannot receive data from or provide data to a relational database.

#### Related reference

“EGL library DateTimeLib” on page 919

“Datetime expressions” on page 591

“Primitive types” on page 34

“Date, time, and timestamp format specifiers” on page 46

## TIME

An item of type TIME is a series of six single-byte numeric digits that reflect a specific moment.

The format of type TIME is *HHmmss*:

*HH*

Two digits that represent the hour. The range is 00 to 24.

*mm*

Two digits that represent the minute within the hour. The range is 00 to 59.

*ss* Two digits that represent the second within the minute. The range is 00 to 59.

The internal hexadecimal representation of an example value is as follows if the item is on a host environment which uses EBCDIC:

```
// 8:40:20 o'clock  
F0 F8 F4 F0 F2 F0
```

The internal hexadecimal representation of an example value is as follows if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// 8:40:20 o'clock  
30 38 34 30 32 30
```

An item of type TIME can receive data from and provide data to a relational database.

### Related reference

“EGL library DateTimeLib” on page 919

“Datetime expressions” on page 591

“Primitive types” on page 34

“Date, time, and timestamp format specifiers” on page 46

## TIMESTAMP

An item of type TIMESTAMP is a series of one to twenty single-byte numeric digits that reflect a specific moment. The meaning of each digit is determined by the mask that you specify when declaring the item.

The following characters are available, in order, when you specify the mask:

*yyyy*

Four digits that represent the year. The range is 0000 to 9999.

*MM*

Two digits that represent the month. The range is 01 to 12.

*dd*

Two digits that represent the day. The range is 01 to 31.

*HH*

Two digits that represent the hour. The range is 00 to 23.

*mm*

Two digits that represent the minute. The range is 00 to 59.

*ss*

Two digits that represent the second. The range is 00 to 59.

*f*

Zero to six digits that each represent a fraction of seconds; the first represents tenths, the second represents hundreds, and so on.

The default mask is *yyyyMMddHHmmss*.

When you interact with DB2 (directly or by way of JDBC) , you must specify every component from year (*yyyy*) through seconds (*ss*). In other contexts, the following is true:

- You can have zero characters of a given kind at the beginning or end of a mask, but cannot skip intermediate characters.
- Valid masks include these:

```
yyyyMMddHHmmss
yyyy
MMss
```

- The following masks are invalid because intermediate characters are missing:

```
// NOT valid
ddMMssffffff
HHssff
```

The internal hexadecimal representation of an example value is as follows if the default mask (*yyyyMMddHHmmss*) is in effect and if the item is on a host environment which uses EBCDIC:

```
// 8:05:10 o'clock on 12 January 2005
F2 F0 F0 F5 F0 F1 F1 F2 F0 F8 F0 F5 F1 F0
```

The internal hexadecimal representation of an example value is as follows if the default mask (*yyyyMMddHHmmss*) is in effect and if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// 8:05:10 o'clock on 12 January 2005
32 30 30 35 30 31 31 32 30 38 30 35 31 30
```

An item of type **TIMESTAMP** can be compared with (or assigned to or from) an item of type **TIMESTAMP** or an item of type **DATE**, **TIME**, **NUM**, or **CHAR**. However, an error occurs at development time if you assign a value that is not valid. An example is as follows:

```
// NOT valid because February 30 is not a valid date
myTS timestamp("yyymdd");
myTS = "20050230";
```

If characters at the beginning of a full mask are missing (for example, if the mask is "dd"), EGL assumes that the higher-level characters ("yyyyMM", in this case) represent the current moment, in accordance with the machine clock. The following statements cause a runtime error in February:

```
// NOT valid because February 30 is not a date
myTS timestamp("dd");
myTS = "30";
```

Finally, an item of type **TIMESTAMP** can receive data from or provide data to a relational database.

### Related reference

"Assignments" on page 456

"Date, time, and timestamp format specifiers" on page 46

"Datetime expressions" on page 591

"EGL library DateTimeLib" on page 919

"Logical expressions" on page 593

"Primitive types" on page 34

## Date, time, and timestamp format specifiers

The formats of dates, times, and timestamps are specified by a pattern of letters, each representing a component of the date or time. These characters are case-sensitive, and all letters from a to z and from A to Z parse to a component of the date or time.

To display letters in the date, time, or timestamp without that text being parsed as a component of the date or time, enclose that letter or letters in single quotes. To display a single quote in the date, time, or timestamp, use two single quotes.

The following table lists the letters and their values in a date, time, or timestamp pattern.

Letter	Date or time component	Type	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	AM/PM marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in AM/PM (0-11)	Number	0
h	Hour in AM/PM (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-800
C	Century	Century	20; 21

The number of each letter used consecutively in the pattern determines how that group of letters is interpreted and parsed. The interpretation depends on the type of letter. Also, the interpretation depends on whether the pattern is being used for formatting or parsing. The following list describes the types of letters and how different numbers of those letters affect the interpretation.

**Text** For formatting, if the number of letters is less than 4, the full form is used. Otherwise, an abbreviation is used, if available. In parsing, both forms are accepted, independent of the number of pattern letters.

### Number

For formatting, the number of pattern letters represents the minimum number of digits. Zeroes are added to shorter numbers to make them the

designated length. For parsing, the number of pattern letters is ignored unless it is needed to separate two adjacent fields.

**Year** For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits. Otherwise, it is interpreted as the Number type.

For parsing, if the number of pattern letters is not 2, the year is interpreted literally, regardless of the number of digits. For example, the pattern MM/dd/yyyy assigned the value 01/11/12 parses to January 11, 12 A.D. The same pattern assigned the value 01/02/3 or 01/02/0003 parses to January 2, 3 A.D. In the same way, the same pattern assigned the value 01/02/-3 parses to January 2, 4 B.C.

For parsing, if the pattern is yy, the parser determines the full year relative to the current year. The parser assumes that the two-digit year is within 80 years before or 20 years after the time of processing. For example, if the current year is 2004, the pattern MM/dd/yy assigned the value 01/11/12 parses to January 11, 2012, while the same pattern assigned the value 05/04/64 parses to May 4, 1964.

### Month

If the number of pattern letters is 3 or more, the month is interpreted as the Text type. Otherwise, it is interpreted as the Number type.

### General time zone

General time zones are interpreted as the Text type if they have names. For time zones representing a GMT offset value, the following syntax is used:

`GMTOffsetTimeZone = GMT Sign Hours : Minutes`

**Sign** Either + or -

**Hours** A one-digit or two-digit number from 0 to 23. The format is locale independent and must be taken from the Basic Latin block of the Unicode standard.

### Minutes

A two-digit number from 00 to 59. The format is locale independent and must be taken from the Basic Latin block of the Unicode standard.

For parsing, RFC 822 time zones are also accepted.

### RFC 822 time zone

For formatting, the RFC 822 4-digit time zone format is used

`RFC822TimeZone = Sign TwoDigitHours : Minutes`

*TwoDigitHours* must be a two-digit number from 00 to 23. The other definitions are the same as the General time zone type.

For parsing, General time zones are also accepted.

### Century

Displayed as a Number type that shows the result of the following calculation: full year divided by 100, with the remainder ignored.

The following table lists some examples of date and time patterns interpreted in the U.S. locale.

Date and Time Pattern	Result
yyyy.MM.dd G 'at' HH:mm:ss z	2001.07.04 AD at 12:08:56 PDT

Date and Time Pattern	Result
EEE, MMM d, 'yy	Wed, Jul 4, '01
h:mm a	12:08 PM
hh 'o''clock' a, zzzz	12 o'clock PM, Pacific Daylight Time
K:mm a, z	0:08 PM, PDT
yyyyy.MMMM.dd GGG hh:mm aaa	02001.July.04 AD 12:08 PM
EEE, d MMM yyyy HH:mm:ss Z	Wed, 4 Jul 2001 12:08:56 -0700
yyMMddHHmmssZ	010704120856-0700

## LOB types

### CLOB

An item of type CLOB represents a character large object with a length ranging from one byte to two gigabytes.

The following statements are true of an item of type CLOB:

- It can be declared only as an individual item, and is not supported in BasicRecords.
- It can be passed to local function and program calls. Large object parameters and corresponding arguments must both be declared as large objects of the same type.
- It can be assigned only to another Clob variable.
- It can be moved to another Clob variable, which has the same result as being assigned to a Clob variable.
- You can create a reference variable of BLOB.
- It uses SQLlocator (CLOB); that is, CLOB contains a logical pointer to the SQL CLOB data rather than to the data itself.
- When used with SQLRecord,
  - CLOB represents Character Large Object as a column in the database.
  - CLOB is valid for the duration of the transaction in which it was created.
- It cannot be passed to calls to remote programs or to non-EGL programs.
- It cannot be referenced as an operand on assignment statements or in expressions.

You may use the following functions with CLOB:

- attachClobToFile
- freeClob
- getClobLen
- getStrFromClob
- getSubStrFromClob
- loadClobFromFile
- setClobFromString
- setClobFromStringAtPosition
- truncateClob

- `updateClobToFile`

#### Related reference

"BLOB"

"EGL library LobLib" on page 958

"attachClobToFile()" on page 960

"freeClob()" on page 961

"getClobLen()" on page 962

"getStrFromClob()" on page 962

"getSubStrFromClob()" on page 962

"loadClobFromFile()" on page 963

"setClobFromString()" on page 963

"setClobFromStringAtPosition()" on page 964

"truncateClob()" on page 965

"updateClobToFile()" on page 965

"Primitive types" on page 34

## BLOB

An item of type BLOB represents a binary large object with a length ranging from one byte to two gigabytes.

The following statements are true of an item of type BLOB:

- It can be declared only as an individual item, and is not supported in BasicRecords.
- It can be passed to local function and program calls. Large object parameters and corresponding arguments must both be declared as large objects of the same type.
- It can be assigned only to another Blob variable.
- It can be moved to another Blob variable, which has the same result as being assigned to a Blob variable.
- You can create a reference variable of BLOB.
- It uses SQLLocator (BLOB); that is, BLOB contains a logical pointer to the SQL BLOB data rather than to the data itself.
- When used with SQLRecord,
  - BLOB represents Binary Large Object as a column in the database.
  - BLOB is valid for the duration of the transaction in which it was created.
- It cannot be passed to calls to remote programs or to non-EGL programs.
- It cannot be referenced as an operand on assignment statements or in expressions.

You may use the following functions with BLOB:

- `attachBlobToFile`
- `freeBlob`
- `getBlobLen`
- `loadBlobFromFile`
- `truncateBlob`
- `updateBlobToFile`

#### Related reference

"CLOB" on page 48

"EGL library LobLib" on page 958



`"attachBlobToFile()"` on page 959  
`"freeBlob()"` on page 961  
`"getBlobLen()"` on page 961  
`"loadBlobFromFile()"` on page 963  
`"truncateBlob()"` on page 964  
`"updateClobToFile()"` on page 965  
`"Primitive types"` on page 34

## Numeric types

### BIN and the integer types

An item of type BIN is interpreted as a binary value. The length can be 4, 9, or 18 and reflects the number of positive digits in decimal format, including any decimal places. The value -12.34, for example, fits in an item of length 4. A 4-digit number requires 2 bytes; a 9-digit number requires 4 bytes; and an 18-digit number requires 8 bytes.

For an item of length 4, the internal bit representations of example values are as follows:

```
// for decimal 1234, the hexadecimal value is 04 D2:  
00000100 11010010  
  
// for decimal -1234, the value is the 2's complement (FB 2E):  
11111011 00101110
```

It is recommended that you use items of type BIN instead of other numeric types whenever possible; for example, for arithmetic operands or results, for array subscripts, and for key items in relative records.

The following types are equivalent to type BIN:

- BIGINT is length 18, no decimal places
- INT is length 9, no decimal places
- SMALLINT is length 4, no decimal places

### Related reference

`"Primitive types"` on page 34

### DECIMAL

An item of type DECIMAL is a numeric value in which each half-byte is a hexadecimal character, and the sign is represented by a hexadecimal C (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte.

The length reflects the number of digits and ranges from 1 to 32.

To determine the number of bytes, add 2 to the length value, divide the sum by 2, and truncate any fraction in the result.

For an item of length 4, the internal hexadecimal representations of example values are as follows:

```
// for decimal 123  
00 12 3C  
  
// for decimal -123  
00 12 3D
```

```
// for decimal 1234
01 23 4C

// for decimal -1234
01 23 4D
```

A negative value that is read from a file or database into a field of type DECIMAL may have a hexadecimal B in place of a D; EGL accepts the value but converts the B to D.

The format of a DB2 UDB column of type DECIMAL is equivalent to the format of a DECIMAL-type host variable.

#### Related reference

“Primitive types” on page 34

## FLOAT

An item of type FLOAT is interpreted as a binary value for double-precision floating-point numbers with as many as 16 significant digits. The length is fixed at 8 bytes. In EGL-generated Java programs, the value ranges from 4.9e-324 to 1.7976931348623157e308.

FLOAT corresponds to each of these definitions:

- The FLOAT data type in a relational database management system
- The **double** data type in C, C++, or Java

For floating-point values, format conversion between Java and host COBOL formats is supported by DB2 but is not supported on calls to host programs.

#### Related reference

“Primitive types” on page 34

## MONEY

An item of type MONEY is a numeric value that is equivalent in most respects to an item of type DECIMAL. In the case of MONEY, the default for length is 16; the default for decimal places is 2; the minimum length is 2; and a currency symbol is displayed in output fields. MONEY corresponds to the IBM Informix 4GL MONEY data type.

The format is based on the variable defaultMoneyFormat.

#### Related reference

“DECIMAL” on page 50

“Formatting properties” on page 67

“Primitive types” on page 34

## NUM

An item of type NUM is a numeric value in which each byte is a digit in character format, and the sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. The length reflects both the number of digits and the number of bytes. The length ranges from 1 to 32.

For an item of length 4, the internal hexadecimal representations of example values are as follows if the item is on a host environment which uses EBCDIC:

```
// for decimal 1234
F1 F2 F3 F4

// for decimal -1234
F1 F2 F3 D4
```

The internal hexadecimal representations of example values are as follows if the item is on a workstation environment like Windows 2000, which uses ASCII:

```
// for decimal 1234
31 32 33 34

// for decimal -1234
31 32 33 74
```

### Related reference

“Primitive types” on page 34

## NUMC

A field of type NUMC is a numeric value in which each byte is a digit in character format, and the sign is represented by a sign-specific hexadecimal value in the left half of the rightmost byte. The length reflects both the number of digits and the number of bytes and ranges from 1 to 18.

For a field of length 4, the internal hexadecimal representations of example values are as follows if the field is on a host environment which uses EBCDIC:

```
// for decimal 1234
F1 F2 F3 C4

// for decimal -1234
F1 F2 F3 D4
```

The internal hexadecimal representations of example values are as follows if the field is on a workstation environment like Windows 2000, which uses ASCII:

```
// for decimal 1234
31 32 33 34

// for decimal -1234
31 32 33 74
```

### Related reference

“Primitive types” on page 34

## PACF

A field of type PACF is a numeric value in which each half-byte is a hexadecimal character, and the sign is represented by a hexadecimal F (for a positive number) or a hexadecimal D (for a negative number) in the right half of the rightmost byte. The length reflects the number of digits and ranges from 1 to 18. To determine the number of bytes, add 2 to the length value, divide the sum by 2, and truncate any fraction in the result.

For a field of length 4, the internal hexadecimal representations of example values are as follows:

```
// for decimal 123
00 12 3F

// for decimal -123
00 12 3D

// for decimal 1234
```

```

01 23 4F
// for decimal -1234
01 23 4D

```

A negative value that is read from a file or database into a field of type PACF may have a hexadecimal B in place of D; EGL accepts the value but converts the B to D.

#### Related reference

“Primitive types” on page 34

## SMALLFLOAT

An item of type SMALLFLOAT is interpreted as a binary value for single-precision floating-point numbers with as many as 8 significant digits. The length is fixed at 4 bytes of memory storage.

In EGL-generated Java programs, the value ranges from 3.40282347e+38 to 1.40239846e-45.

SMALLFLOAT corresponds to each of these definitions:

- The SMALLFLOAT data type in a relational database management system
- The **float** data type in C, C++, or Java

For floating-point values, format conversion between Java and host COBOL formats is supported by DB2 but is not supported on calls to host programs.

#### Related reference

“Primitive types” on page 34

---

## Declaring variables and constants in EGL

You can declare a variable in these ways:

- You can base a variable on one of several primitive types, as in this example--  
myItem CHAR(10);
- You can base a variable on a DataItem part or record part, as in this example--  
myRecord myRecordPart;
- You can base a variable on one of the predefined parts, as in this example of a dictionary--

```

myVariable Dictionary
{
    empnum=0005,
    lastName="Twain",
    firstName="Mark",
    birthday="021460"
};

```

- You can declare a record that redefines the area of memory declared by another record; for details, see *Declaring a record that redefines another*.
- A program or other generatable part can access the fields of a DataTable, which is treated as a variable that is global to either the program or the run unit. You can use a simpler syntax to access those fields if the DataTable is listed in one of the program’s use declarations.
- A program can access the fields of a text or print form, which is treated as a variable that is global to the program. The program must include the related formGroup in a use declaration.

- A program or other generatable logic part can access the library variables that are declared outside of any library function. Those variables are global to the run unit. You can use a simpler syntax to access those fields if the library is listed in one of the program's use declarations.

You declare a constant by specifying the symbol `CONST` followed by the constant name, type, equal sign, and value; and the specified value cannot be changed at run time. Examples are as follows:

```
const myString String = "Great software!";
const myArray BIN[] = [36, 49, 64];
const myArray02 BIN[] [] = [[1,2,3],[5,6,7]];
```

A constant cannot be in a record or other complex structure.

Finally, to declare multiple variables or constants in a single statement, separate one identifier from the next by a comma, as in these examples:

```
const myString01, myString02 STRING = "INITIAL";
myItem01, myItem02, myItem03 CHAR(5);
myRecord01, myRecord02 myRecordPart;
```

#### Related concepts

"References to parts" on page 23

"Parts" on page 19

"Typedef" on page 28

#### Related tasks

"Declaring a record that redefines another"

#### Related reference

"Primitive types" on page 34

"Use declaration" on page 1091

## Declaring a record that redefines another

You may wish to declare a record that redefines an area of memory declared by another record. For example, you can write a loop that reads one data area after another from a serial file, when the structure of the retrieved data is different from one retrieval to the next, as in the following example:

```
Record RecordA type SerialRecord
{ fileName = "myFile" }
record_type char(1);
item1 char(20);
end

Record RecordB type BasicRecord
10 record_type char(1);
10 item2 bigint;
10 item3 decimal(7);
10 item4 char(8);
end

Program ProgramX type basicProgram
myRecordA RecordA;
myRecordB RecordB {redefines = "myRecordA"};

function main();
get myRecordA;
while (myRecordA not endOfFile)
if (myRecordA.record_type == "A")
myFunction01(myRecordA.item1);
```

```

        else
            myFunction02(myRecordB.item2, myRecordB.item3, myRecordB.item4);
        end
        get myRecordA;
    end
end
end

```

Within the loop, the function acts as follows:

1. Checks the first field in the input record for a code that identifies how the rest of the data is structured.
2. Processes the other fields in the retrieved data by using either the input record or a second, basic record. The second record refers to the same area of memory as the input record but is structured differently.

To declare one record as a redefinition of another, you use the property **redefines**, which accepts a string that identifies another record. This property is only available in a record declaration, not in a record part definition.

The original and overlay record can be any types of fixed record, with the following restrictions:

- The records must be in the same scope. If one record is declared as a field of a non-fixed record, for example, the other record must be declared as a field in the same non-fixed record. Similarly, if one record is declared in a library but outside of a function, the other must be declared in the same library but outside of a function.
- The overlay record must have the same length or be shorter than the original record. This restriction prevents your code from accessing an area in the overlay record that is not in the area of memory being redefined.
- Use of an SQL record (or any record that is nullable) must take into account the hidden bytes that are described in *SQL record internals*.

An overlay record does not have any of the information (other than structure) that is in the original record. An indexed record can redefine a serial record, for example, but the file accessed by the indexed record is identified in the indexed record part and not in the serial record part.

#### Related concepts

“References to variables in EGL” on page 59

#### Related concepts

“Declaring variables and constants in EGL” on page 53

#### Related reference

“SQL record internals” on page 876

---

## Dynamic and static access

EGL resolves a variable reference by static or dynamic access:

- When *dynamic access* is in effect, the field name and type are known only at run time. Your code determines the name from a value in the code or from runtime input.

Dynamic access is in effect when your code is referencing any of these:

- A variable whose primitive type is ANY.
- A value field in a dictionary; that field is of type ANY.

- A field in a record, when the chain of relationships that led to that field (from record to field to subfield) is such that a previous reference used dynamic access.
- A field that is referenced by the EGL bracket syntax. In this case, the field name does not necessarily follow the rules for identifiers, but can be an EGL reserved word or can include spaces and other characters that would not be valid otherwise.

For details, see *Bracket syntax for dynamic access*.

- When *static access* is in effect, the field name and type are known at generation time, and the name is always consistent with the naming conventions for EGL identifiers. The name is not used at run time.

Static access is in effect when your code is referencing any of these:

- A variable that is outside of any container and whose type is other than ANY
- A field in a fixed record
- A field in a non-fixed record, when the chain of relationships that led to that field (from variable to field to subfield) is such that every reference used static access

Consider an example in which the values in a dictionary include a fixed record and a non-fixed record:

```
// a fixed record part
Record myFixedRecordPart type=serialRecord
{
    fileName = "myFile"
}
10 ID INT;
10 Job CHAR(10);
end

// a record part (not fixed)
Record myDynamicRecordPart type=basicRecord
ID INT;
Job CHAR(10);
end

Program myProgram

dynamicPerson myDynamicRecordPart;
myFlexID INT;

fixedPerson myFixedRecordPart;
myFixedID INT;

Function main()

dynamicPerson.ID = 123;
dynamicPerson.Job = "Student";

fixedPerson.ID = 456;
fixedPerson.Job = "Teacher";

relationship Dictionary
{
    dynamicRecord=dynamicPerson,
    staticRecord=fixedPerson
};
end
end
end
```



The following rules apply:

- A reference to a dictionary value is dynamic and every subordinate reference is dynamic. Consider the effect if the code included the following statements:

```
myDynamicID INT;  
myDynamicID = relationship.dynamicRecord.ID;
```

The reference to `dynamicRecord` would be dynamic, and the reference to `ID` would be dynamic, with the identifier `ID` visible at run time.

- A reference that begins with a fixed structure can reference only the memory internal to that structure. In the current example, a reference that begins with `fixedPerson` can access the fields `ID` and `JOB` in the fixed record but can access no other fields.
- Your code can access a fixed structure dynamically but the same reference statement cannot access the fields of that field. In the current example, the following reference would not be valid because the identifier `ID` is not available at run time:

```
myFixedID INT;  
  
// NOT valid  
myFixedID = relationship.fixedRecord.ID;
```

You could handle the problem by declaring another fixed record and assigning it values from the fixed record that is in the dictionary:

```
myFixedID INT;  
myOtherRecord myFixedRecordPart;  
myOtherRecord = relationship.staticRecord;  
myFixedID = myOtherRecord.ID;
```

Dynamic access is valid in assignments (on the left- or right-hand sides); in logical expressions; and in the statements **set**, **for**, and **openUI**.

### Related concepts

"Bracket syntax for dynamic access" on page 61

"Dictionary" on page 83

"Program part" on page 148

"References to variables in EGL" on page 59

"Typedef" on page 28

### Related tasks

"Declaring variables and constants in EGL" on page 53

### Related reference

"Assignments" on page 456

"Logical expressions" on page 593

"Primitive types" on page 34

"set" on page 742

---

## Scoping rules and "this" in EGL

If an EGL part declares a variable or constant, the identifier used in the declaration is *in scope* (available) throughout the part:

- If the declaration is in a function, the identifier is in the local scope of the function. If the function `Function01` declares the variable `Var01`, for example, any code in `Function01` can reference `Var01`. The identifier is available even in function code that precedes the declaration.

The variable can be passed as an argument to another function, but the original identifier is not available in that function. The parameter name is available in the receiving function because the parameter name was declared there.

- If the declaration is in a generatable part such as a program but is outside of any function, the identifier is in *program-global scope*, which means that the identifier can be referenced by any function invoked by that part. For example, if a program declares Var01 and invokes Function01 which in turn invokes Function02, Var01 is available throughout both functions.

The identifiers in a text or print form are global to the generatable part that references the form. Those identifiers are available even in functions that precede the function which presents the form.

- If the declaration is in a library but outside of any function, the identifier is in *run-unit scope*, which means global to all code in the run unit.
- The names of a dataTable and its fields may be in program-global, run-unit, or an even larger scope, depending on the setting of dataTable properties and on the environment in which the dataTable resides.

Identifiers that are identical cannot be in the same scope. However, most identifiers refer to an area of memory that is logically inside a container such as a record; and in those cases your code qualifies an identifier with the name of the enclosing container. If the function variable myString is in a record called myRecord01, for example, your code refers to the variable as a field of the record:

```
myRecord01.myString
```

If the same identifier is in two scopes, any reference to the identifier is a reference to the most local scope, but you can use qualifiers to override that behavior:

- Consider the case of a program that declares variable Var01 and invokes a function that itself declares a variable of the same name. An unqualified reference to Var01 in the function causes access of the locally declared variable. To access an identifier that is program-global even when a local identifier takes precedence, qualify the identifier with the keyword *this*, as in the following example:

```
this.Var01
```

In rare cases the keyword *this* is also used to override a behavior of a set value block in an assignment statement. For details, see *Set value blocks*.

- Consider the following case--
  - A program has a use declaration to access a library; and
  - The program and the library each declare a variable named Var01.

If a function in the program includes an unqualified reference to Var01, the function accesses the program variable.

To access an identifier in run-unit scope even when another identifier prevents that access, qualify the identifier with the part name, as in the following example (where myLib is the name of a library):

```
myLib.Var01
```

If the library or dataTable is in a different package and you have not referenced the part in an import statement, you must preface the part name with the package name, as in the following example (where myPkg is the package name):

```
myPkg.myLib.Var01
```

The package name always qualifies a part name and cannot immediately precede a variable or constant identifier.

Finally, a local identifier may be the same as a dataTable or library name if the local identifier is in a different package from where the dataTable or library resides. To reference the dataTable or library name, include the package name.

#### Related concepts

- "Function part" on page 150
- "Library part of type basicLibrary" on page 169
- "Library part of type basicLibrary" on page 169
- "PageHandler" on page 223
- "Parts" on page 19
- "Program part" on page 148
- "References to parts" on page 23
- "References to variables in EGL"
- "Overview of EGL properties" on page 64
- "Fixed structure" on page 27
- "Typedef" on page 28

#### Related tasks

- "Declaring variables and constants in EGL" on page 53

#### Related reference

- "Function invocations" on page 613
- "Function part in EGL source format" on page 621

---

## References to variables in EGL

For details on the distinction between two kinds of memory access, see *Dynamic and static access*.

Regardless of which kind of access is in effect, the EGL dotted syntax is usually sufficient. Consider the following part definitions, for example:

```
Record myRecordPart01 type basicRecord
  myString      STRING;
  myRecordVar02 myRecordPart02;
end
```

```
Record myRecordPart02 type basicRecord
  myString02    STRING;
  myRecordVar03 myRecordPart03;
  myDictionary  Dictionary
  {
    empnum=0005,
    lastName="Twain",
    firstName="Mark",
    birthday="021460"
  };
end
```

```
Record myRecordPart03 type basicRecord
  myInt INT;
  myDictionary  Dictionary
  {
    customerNum=0005,
    lastName="Clemens"
  };
end
```

Assume that a function uses the record part *myRecordPart01* as the type when declaring a variable named *myRecordVar01*.

To refer to the field `myInt`, list the following symbols in order:

- The name of the variable; in this case, `myRecordVar01`
- A period (`.`)
- A list of the fields that lead to the field of interest, with a period separating one identifier from the next; for example, `myRecordVar02.myRecordVar03`
- The field name of interest, preceded by a period; in this case, `.myInt`

The presence of an array causes a straightforward extension of the same syntax. If `myRecordVar03` were declared as an array of three records, for example, you could use the following symbols to access the field `myInt` in the third element of that array:

```
myRecordVar01.myRecordVar02.myRecordVar03[3].myInt
```

The dotted syntax also works when you reference a dictionary field in this example. To access the value "Twain", specify the following characters on the right-hand side of an assignment statement:

```
myRecordVar01.myRecordVar02.myDictionary.lastName
```

The presence of a field named `myDictionary` in two different record parts does not pose a problem because each same-named field is referenced in relation to its own, enclosing record.

You also can use dotted syntax to refer to a constant (such as `myConst`) in a library (such as `myLib`):

```
myLib.myConstant
```

Two other syntaxes are available:

- When using dynamic access, you may wish to specify a field name as a quoted string or as an identifier of type `STRING`. This capability is used primarily when you are adding or retrieving a dictionary entry (a key-and-value pair), in these cases:
  - The key is an EGL reserved word or includes a character (such as a period or space) that is not valid in an identifier; or
  - You wish to use a string constant to assign or reference the key.

The syntax requires that you place the variable, constant, or literal inside a pair of hard brackets( `[ ]` ). The content-filled brackets are equivalent to a dot followed by a valid identifier, and you can mix the two syntaxes. However, the beginning of a reference must be an identifier.

For examples, see *Bracket syntax for dynamic access*.

- You may want the convenience of an abbreviated syntax for referencing a field in a fixed structure (a `dataTable`, text form, print form, or fixed record). It is recommended that you avoid this syntax, however, in favor of the full qualification described earlier.

An abbreviated syntax can be valid in relation to fixed structures only if you set the property **`allowUnqualifiedItemReferences`** to *yes*. That property is a characteristic of generatable logic parts like programs, libraries, and pageHandlers; and the default value is *no*.

For details, see *Abbreviated syntax for static access*.

### Related concepts

"Abbreviated syntax for referencing fixed structures" on page 62

"Bracket syntax for dynamic access" on page 61

"Dynamic and static access" on page 55  
"Enumerations in EGL" on page 578  
"Function part" on page 150  
"Parts" on page 19  
"Program part" on page 148  
"References to parts" on page 23  
"Scoping rules and "this" in EGL" on page 57  
"Fixed structure" on page 27  
"Typedef" on page 28

#### **Related tasks**

"Declaring variables and constants in EGL" on page 53

#### **Related reference**

"Arrays" on page 75  
"Function invocations" on page 613  
"Function part in EGL source format" on page 621  
"Primitive types" on page 34  
"Use declaration" on page 1091

## **Bracket syntax for dynamic access**

Wherever dynamic access is valid, you can reference a field by using a string variable, constant, or literal in brackets. Each content-filled pair of brackets is equivalent to a dot followed by a valid identifier.

Although any keys specified in a dictionary declaration must fulfill the rules for EGL identifiers, you can specify a wider range of keys by using bracket syntax in EGL assignment statements. Bracket syntax is required in the next example, where two entries are added to a dictionary and the value in each of those entries is retrieved:

```
row Dictionary { lastname = "Smith" };
category, motto STRING;

row["Record"] ="Reserved word";
row["ibm.com"]="Think!";

category = row["Record"];
motto    = row["ibm.com"]
```

If you reference a value by using an identifier in dotted syntax, you can reference the same value in bracket syntax by using a string that is equivalent to the identifier. The following assignments have the same effect:

```
row.age = 20;
row["age"] = 20;
```

Assume that you declared a record named myRecordVar01, which includes a field named myRecordVar02, and that myRecordVar02 is itself a record that includes the previous dictionary. A valid reference is as follows:

```
myRecordVar01.myRecordVar02.row.lastName
```

Access is static for most of that reference. Dynamic access begins when you access the field in the dictionary. Assume that these constants are in scope, however:

```
const SECOND STRING = "myRecordVar02";
const GROUP  STRING = "row";
const LAST   STRING = "lastName";
```

You can code the previous reference as follows:

```
myRecordVar01[SECOND][GROUP][LAST]
```

The first symbol in a reference must always be a valid identifier, but in this case, dynamic access is in effect after that identifier.

You can mix the dotted and bracket syntaxes. For example, the following reference is equivalent to the previous one:

```
myRecordVar01[SECOND].row[LAST]
```

As a final example, consider a reference with an array index:

```
myRecordVar01.myRecordVar02.myRecordVar03[3][2].myInt
```

Assume that these constants are in scope:

```
const SECOND  STRING = "myRecordVar02";  
const THIRD   STRING = "myRecordVar03";  
const CONTENT STRING = "myInt";
```

You can code the previous reference in these ways:

```
myRecordVar01[SECOND][THIRD][3][2][CONTENT]
```

```
myRecordVar01[SECOND][THIRD][3][2].myInt
```

```
myRecordVar01.myRecordVar02.THIRD[3][2][CONTENT]
```

### Related concepts

“Abbreviated syntax for referencing fixed structures”

“Dynamic and static access” on page 55

“Function part” on page 150

“Parts” on page 19

“Program part” on page 148

“References to parts” on page 23

“References to variables in EGL” on page 59

“Scoping rules and “this” in EGL” on page 57

“Fixed structure” on page 27

“Typedef” on page 28

### Related tasks

“Declaring variables and constants in EGL” on page 53

### Related reference

“Arrays” on page 75

“Function invocations” on page 613

“Function part in EGL source format” on page 621

“Options records for MQ records” on page 772

“Primitive types” on page 34

“Use declaration” on page 1091

## Abbreviated syntax for referencing fixed structures

The following rules are in effect for referencing fields in a dataTable, text form, print form, or fixed record:

- If you are referencing a field in a container such as a fixed record, you can use the usual dotted syntax to avoid ambiguity about the area of memory being referenced. Consider the following part declaration, for example:

```

Record myRecordPart type serialRecord
{
    fileName = "myFile"
}
10 myTop;
20 myNext;
30 myAlmost;
40 myChar CHAR(10);
40 myChar02 CHAR(10);
end

```

Assume that a function uses the record part *myRecordPart* as the type when declaring a variable named *myRecordVar*.

A valid reference to *myChar* in *myRecordVar* is as follows:

```
myRecordVar.myTop.myNext.myAlmost.myChar
```

That reference is considered to be *fully qualified*.

- If you want to refer to a field whose name is unique within a structure, you can specify the variable name, followed by a period, followed by the field name. Valid references for the earlier example include this symbol:

```
myRecordVar.myChar
```

That reference is considered to be *partially qualified*.

You cannot partially qualify a field name in any other way. You cannot include only some of the field names that are between the variable name and the field name of interest, for example, nor can you eliminate the variable name while keeping any of the names of structure field that are superior to the field of interest. The following references are *not* valid for the earlier example:

```

// NOT valid
myRecordVar.myNext.myChar
myRecordVar.myAlmost.myChar
myNext.myChar
myAlmost.myChar

```

- You can refer to a field without preceding the name with any qualifiers. Valid references for the earlier example include these symbols:

```
myChar
myChar02
```

Those references are considered to be *unqualified*.

- You must qualify any reference to a structure field to the extent necessary to avoid ambiguity.
- The name of a structure field can be an asterisk (\*) if the related memory area is a *filler*, which is an area whose name is of no importance. You cannot include an asterisk in a reference. Consider this example:

```

record myRecordPart type serialRecord
{
    fileName = "myFile"
}
10 person;
20 *;
30 streetAddress1 CHAR(30);
30 streetAddress2 CHAR(30);
30 nation CHAR(20);
end

```

If you use that part as a type when declaring the variable *myRecordVar*, you can refer to *myRecordVar.nation* or *nation*, but the following references are not valid:

```

// NOT valid
myRecordVar.*.streetAddress1
myRecordVar.*.streetAddress2
myRecordVar.*.nation

```



- When EGL tries to resolve a reference, names of local variables are searched first, then names of structure fields in the records used for I/O in the same function, then names of other local structure fields, then names that are program-global.

Consider the case in which a function declares both a primitive variable called *nation* and a variable that points to the following basic record:

```
record myRecordPart
  10 myTop;
  20 myNext;
  30 nation CHAR(20);
end
```

An unqualified reference to *nation* refers to the primitive variable, not to the structure field.

- A name search shows no preference for program-global primitive variables over program-global structure fields. Consider the case in which a program declares both a primitive variable called *nation* and a variable that points to the format of the following basic record:

```
record myRecordPart
  10 myTop;
  20 myNext;
  30 nation CHAR(20);
end
```

An unqualified reference to *nation* fails because *nation* could refer either to the primitive variable or to the structure field. You can reference the structure field, but only by qualifying the reference.

For additional rules, see *Arrays* and *Use declaration*.

### Related concepts

"Function part" on page 150

"Parts" on page 19

"Program part" on page 148

"References to parts" on page 23

"References to variables in EGL" on page 59

"Scoping rules and "this" in EGL" on page 57

"Fixed structure" on page 27

"Typedef" on page 28

### Related tasks

"Declaring variables and constants in EGL" on page 53

### Related reference

"Arrays" on page 75

"Function invocations" on page 613

"Function part in EGL source format" on page 621

"Options records for MQ records" on page 772

"Primitive types" on page 34

"Use declaration" on page 1091

---

## Overview of EGL properties

Most EGL parts have a set of properties that are used to create appropriate output at generation time. You set those properties in a *set-value block*, which is an area of code that is shown later in the current topic and is described more fully in *Set-value blocks*.

The set of valid properties varies by context:

- Each type of part defines a set of properties; you can change those properties to modify characteristics of the part. Each program part, for example, has a property called **alias** that identifies the name of the compilable unit.

If a part is itself a subtype, additional properties are available. A program of type **textUI** has a property called **alias**, as well as a property called **inputForm**. The latter identifies a text form that is presented to the user before the program logic runs.

- Many part types also define a set of properties for use in any of the primitive fields that are components of that part type. A record part of type **SQLRecord**, for example, includes a set of primitive fields, and each has a **column** property that identifies the SQL table column accessed by the field.

The properties available in a **DataItem** part include *all* the primitive field-level properties that are valid in any context. Consider, for example, a **DataItem** part that represents an ID of nine (and only nine) digits, where in some cases the ID is associated with a relational-database column called **SSN**:

```
DataItem IDPart CHAR(9)
{
  minInput = 9,      // requires 9 input characters
  isDigits = yes,    // requires digits
  columnName = "SSN" // is related to a column
}
```

You can declare a variable of type **IDPart** as follows:

```
myVariable IDPart;
```

You can declare that variable in a composite part such as a record part or directly in a logic part such as program. In every case, the part type determines whether a given property is used.

In the current example, the property **columnName** is used only if the variable is declared in a record of type **SQLRecord**. The two validation properties are used only if the variable is declared in a user-interface part such as a **PageHandler**.

- In some variable declarations, you can override a property that was specified in the related part definition, but only if the property is useful in the context in which the variable is declared:
  - Overriding in context is possible when you declare a variable that is based on a **DataItem** part. The following statement declares a **PageHandler** field of type **SSN** (as defined earlier), but the statement does not require that the user type digits:

```
myVariable IDPart { isDigits = no };
```

In this example, the property **minInput** is unaffected by the override, and the property **columnName** is ignored.
  - In most cases, overrides are not possible for properties of composite parts such as **Record** parts.
- When you define a fixed structure, you can assign properties to the elementary structure fields and can override those properties when you declare a related variable. You also may assign properties to a structure field that has subordinate structure fields, but in those cases, the assigned properties are ignored unless the property documentation says otherwise.
- When you declare a variable of a primitive type, you can set any of the primitive field-level properties that are useful in the context of the variable declaration.
- When you declare a record, you can assign the **redefines** property, which cannot be set when you define a part. For details on this exception, see *Declaring a record that redefines another*.

A property cannot be accessed at run time. When you create variables that are based on an SQL record part, for example, the logic that you write cannot retrieve or change the names assigned to the **tableNames** property, which identifies the SQL tables that are accessed by the record. Even if you override a property value in a variable declaration, the value that you specify at development time cannot be changed by your logic.

The lack of runtime access to a property value means that when you assign the content of a variable or use the variable as an argument, the property value is not transferred along with the content. If you copy data from one SQL record to another, for example, no change is made to the specification of which SQL tables are accessed by the destination record. Similarly, when you pass an SQL record to an EGL function, the parameter receives field content, but retains the SQL-table specifications that were assigned at development time.

Predefined EGL parts such as `ConsoleField` may include both properties and predefined fields. Unlike properties, the fields (sometimes called *attributes*) are available at run time. The logic that you write can read the field value and in many cases can change the field value.

You access predefined fields as you access the fields that you define. You can use a set-value block to set fields as well as to set properties.

**Note:** A restriction applies to fields in fixed structures. You can use set-value blocks to assign the values of the primitive field-level properties, but not to set the values of the fields themselves.

If you make multiple assignments to the same field or property in a set-value block, the last assignment takes effect. That rule also applies to fields of complex properties, which are described in the next section.

## Complex properties

In some cases you specify generation characteristics by assigning complex properties, each composed of a set of property fields. You might use syntax like the following, for example, to declare an EGL service and to define the complex property `@EGLBinding`, which contains the details needed to provide access to the service:

```
myService myServicePart
{ @EGLBinding
  {commType="DIRECT",
   serviceName="myService",
   servicePackage="my.useful.service"}
}
```

Neither a complex property nor its fields can be accessed at run time.

### Related concepts

“References to variables in EGL” on page 59

“Set-value blocks” on page 68

### Related concepts

“References to variables in EGL” on page 59

“Set-value blocks” on page 68

### Related tasks

“Declaring a record that redefines another” on page 54

#### Related reference

“Form part in EGL source format” on page 606  
“Primitive field-level properties” on page 793

## Field-presentation properties

The EGL field-presentation properties specify characteristics that are meaningful when a field is displayed in an on-screen output, when the destination is a command window, but not a Web browser.

The properties are as follows:

- “color” on page 802
- “highlight” on page 811
- “intensity” on page 812
- “outline” on page 820

In addition, the following properties are meaningful when the field is displayed in a printable output, when the destination is a printer or a print file:

- **highlight** property (but only for *underline* and *noHighLight*)
- **outline** property, which is appropriate only for devices that support double-byte characters

The field-presentation properties have no effect on data that is returned to the program from a text form; they are solely for output.

## Formatting properties

The formatting properties specify characteristics that are meaningful when data is presented on a form or a Web browser:

- “align” on page 801
- “currency” on page 804
- “currencySymbol” on page 805
- “dateFormat” on page 805
- “fillCharacter” on page 810
- “isBoolean” on page 812
- “lineWrap” on page 815
- “lowerCase” on page 815
- “masked” on page 816
- “numericSeparator” on page 820
- “outline” on page 820
- “sign” on page 825
- “timeFormat” on page 827
- “timestampFormat” on page 828
- “upperCase” on page 831
- “zeroFormat” on page 837

#### Related concepts

“Overview of EGL properties” on page 64

## SQL item properties

The SQL item properties specify characteristics that are meaningful when an item is used in a record of type `SQLRecord`. You do not need to specify any of the SQL item properties, however, as default values are available.

The properties are as follows:

- “column” on page 803
- “isNullable” on page 813
- “isReadOnly” on page 814
- “maxLen” on page 816
- “persistent” on page 821
- “sqlDataCode” on page 825
- “sqlVariableLen” on page 826

## Validation properties

The validation properties restrict what is accepted when the user enters data in a text form or Web page.

The properties are as follows:

- “fill” on page 810
- “inputRequired” on page 811
- “inputRequiredMsgKey” on page 811
- “isDecimalDigit” on page 813
- “isHexDigit” on page 813
- “minimumInput” on page 816
- “minimumInputMsgKey” on page 817
- “needsSOSI” on page 818
- “typeChkMsgKey” on page 829
- “validatorDataTable” on page 832
- “validatorDataTableMsgKey” on page 833
- “validatorFunction” on page 833
- “validatorFunctionMsgKey” on page 834
- “validValues” on page 834
- “validValuesMsgKey” on page 836

---

## Set-value blocks

A set-value block is an area of code in which you can set both property and field values. For background, see *Overview of EGL properties*.

A set-value block is available when you take any of the following actions:

- Define a part
- Declare a variable
- Code a special form of an assignment statement
- Code an **openUI** statement, as described in *openUI*

In the last two cases, you can assign values only to fields.

**Note:** A restriction applies to fields in fixed structures. You can use set-value blocks to assign the values of the primitive field-level properties, but not to set the values of the fields themselves.

## Set-value blocks for elementary situations

Consider the rules that apply in the most elementary cases:

- Each set-value block begins with a left curly brace (`{`), includes either a list of entries that are separated by commas or a single entry, and ends with a right curly brace (`}`)
- The entries are all in one of two formats:
  - Each entry is composed of an identifier-and-value pair such as **inputRequired** = **yes**; or
  - Each entry contains values that are assigned positionally, when successive values are assigned to successive elements of an array.

In all cases, the set-value block is in the scope of the part, variable, or field being modified. The variations in syntax are best illustrated by example.

The first example shows a `dataItem` part, which has two properties (**inputRequired** and **align**) :

```
// the scope of the set-value block is myPart
DataItem myPart INT
{
  inputRequired = yes,
  align = left
}
end
```

The next example shows a variable of primitive type.

```
// the scope is myVariable
myVariable INT
{
  inputRequired = yes,
  align = left
};
```

The next example shows an SQL record part declaration, which includes two record properties (**tableNames** and **keyItems**):

```
// The scope is myRecordPart
Record myRecordPart type SQLRecord
{ tableNames = ["myTable"],
  keyItems = ["myKey"] }
myKey CHAR(10);
myOtherKey CHAR(10);
myContent01 CHAR(60);
myContent02 CHAR(60);
end
```

The next example shows a variable declaration that uses the previous part as a type, overrides one of the two record properties, and sets two fields in the record:

```
// The scope is myRecord
myRecord myRecordPart
{
  keyItems = ["myOtherKey"],
  myContent01 = "abc",
  myContent02 = "xyz"
};
```

Additional examples include variable declarations and assignment statements:

```
// the example shows the only case in which a
// record property can be overridden in a
// variable declaration.
// the scope is myRecord
myRecord myRecordPart {keyItems = ["myOtherKey"]};

// the scope is myInteger, which is an array
myInteger INT[5] {1,2,3,4,5};

// these assignment statements
// have no set-value blocks
myRecord02.myContent01 = "abc";
myRecord02.myContent02 = "xyz";

// this abbreviated assignment statement
// is equivalent to the previous two, and
// the scope is myRecord02
myRecord02
{
    myContent01="abc",
    myContent02="xyz"
};

// This abbreviated assignment statement
// resets the first four elements of the array
// declared earlier
myInteger{6,7,8,9};
```

The abbreviated assignment statement is not available for fields in a fixed structure.

## Set-value blocks for a field of a field

When you are assigning values for a field of a field, you use a syntax in which the set-value block is in a scope such that the entries are modifying only the field of interest.

Consider the following part definitions:

```
record myBasicRecPart03 type basicRecord
    myInt04 INT;
end

record myBasicRecPart02 type basicRecord
    myInt03 INT;
    myRec03 myBasicRecPart03;
end

record myBasicRecPart type basicRecord
    myInt01 INT;
    myInt02 INT;
    myRec02 myBasicRecPart02;
end
```

You can assign a property value for any field as follows:

- Create a set-value block for the record
- Embed a series of field names to narrow the scope
- Create the field-specific set-value block

The syntax for assigning a property value may take any of three forms, as shown in the following examples, which apply to the field myInt04:



```

// dotted syntax, as described in
// References to variables in EGL.
myRecB myBasicRecPart
{
    myRec02.myRec03.myInt04{ align = left }
};

// bracket syntax, as described in
// Bracket syntax for dynamic access.
// You cannot use this syntax to affect
// fields in fixed structures.
myRecC myBasicRecPart
{
    myRec02["myRec03"]["myInt04">{ align = left }
};

// curly-brace syntax
myRecA myBasicRecPart
{
    myRec02 {myRec03 { myInt04 { align = left }}}
};

```

Even in complex cases, you use a comma to separate one entry in a set-value block from the next; but you need to consider the level at which a given block is nested:

```

// dotted syntax
myRecB myBasicRecPart
{
    myInt01 = 4,
    myInt02 = 5,
    myRec02.myRec03.myInt04{ align = left },
    myRec02.myInt03 = 6
};

// bracket syntax
myRecC myBasicRecPart
{
    myInt01 = 4,
    myInt02 = 5,
    myRec02["myRec03"]["myInt04">{ align = left },
    myRec02["myInt03"] = 6
};

// curly-brace syntax;
// but this usage is much harder to maintain
myRecA myBasicRecPart
{
    myInt01 = 4,
    myInt02 = 5,
    myRec02
    {
        myRec03
        { myInt04
          { action = label5 }},
        myInt03 = 6
    }
};

```

## Use of "this"

In a variable declaration or assignment statement, you can have a container (such as an SQL record) that includes a field (such as *keyItems*) which is named the same as a record property. To refer to your field rather than to the property, use the keyword **this**, which establishes the correct scope for the set-value block or for an entry in the set-value block.

Consider the following record declaration:

```
Record myRecordPart type SQLRecord
{ tableNames = ["myTable"],
  keyItems = ["myKey"] }
myKey CHAR(10);
myOtherKey CHAR(10);
keyItems CHAR(60);
end
```

The following record declaration first sets a value for the property `keyItems`, then sets a value for the field of the same name:

```
myRecord myRecordPart
{
  keyItems = ["myOtherKey"],
  this.keyItems = "abc"
};
```

The next section gives an additional example in an array declaration.

## Set-value blocks, arrays, and array elements

When you declare a dynamic array, you can specify the initial number of elements, as in this example:

```
col1 ConsoleField[5];
```

Assignments in a set-value block refer to properties and predefined fields in each of the initial elements of type `ConsoleField`, though not to any elements that are added later:

```
col1 ConsoleField[5]
{
  position = [1,1],
  color = red
};
```

To assign values to a particular element in a variable declaration, create an embedded set-value block whose scope is that element. As shown in the following example, you specify the scope by using the keyword **this** with a bracketed index:

```
// assign values to the second and fourth element
col1 ConsoleField[5]
{
  this[2] { color = blue },
  this[4] { color = blue }
};
```

For details on another use of the keyword **this**, see *Scoping rules and "this" in EGL*.

You can use positional entries in a set-value block to assign value to successive elements in an array of any of these types (as is relevant only when processing reports or creating console forms):

- `ConsoleField`
- `Menu`
- `MenuItem`
- `Prompt`
- `Report`
- `ReportData`

The following example could be in an OpenUI statement. The scope of each embedded set-value block is a specific array element:

```
new Menu
{
  labelText = "Universe",
  MenuItems =

  // property value is a dynamic array
  [
    new MenuItem
    { name = "Expand",
      labelText = "Expand" },
    new MenuItem
    { name = "Collapse",
      labelText = "Collapse" }
  ]
}
```

## Example with a complex property

To review a syntax that is an extension of what has gone before, consider the complex property **@ProgramLinkData**. That property includes the property fields **programName** and **linkParms**; and **linkParms** takes an array of property fields, each of complex property **@LinkParameter**.

You can define a DataItem part as follows:

```
DataItem Prog1LinkItem char(9)
{ @ProgramLinkData
  {
    programName = "my.company.sys1.PROG1",
    linkParms =
    [ @LinkParameter {name="parm1", value="abc"},
      @LinkParameter {name="parm2",value="efg"} ]
  }
}
```

You can add another property after the set-value block that defines **@ProgramLinkData**, but first, you must add a comma to separate the two properties:

```
DataItem Prog1LinkItem char(9)
{ @ProgramLinkData
  {
    programName = "my.company.sys1.PROG1",
    linkParms =
    [ @LinkParameter {name="parm1", value="abc"},
      @LinkParameter {name="parm2",value="efg"} ]
  },
  displayName = "Go to Program02"
}
```

## Additional examples

Consider the following parts:

```
Record Point
  x, y INT;
end

Record Rectangle
  topLeft, bottomRight Point;
end
```

The following code is valid:

```
Function test()
  screen Rectangle
  {
    topLeft{x=1, y=1},
    bottomRight{x=80, y=24}
  };

  // change x, y in code, using a statement
  // that is equivalent to the following code:
  //   screen.topLeft.x = 1;
  //   screen.topLeft.y = 2;
  screen.topLeft{x=1, y=2};
end
```

Next, initialize a dynamic array of elements of type Point in the same function:

```
pts Point[2]
{
  this[1]{x=1, y=2},
  this[2]{x=2, y=3}
};
```

Set the value of each element that is now in the array, then set the first element to a different value:

```
pts{ x=1, y=1 };
pts[1]{x=10, y=20};
```

In the previous example, `pts[1]` is used rather than `this[1]` because the array name is unambiguous.

Next, consider another dynamic array of type Point:

```
points Point[];
```

The following assignment statement has no effect because no elements exist:

```
points{x=1, y=1};
```

In contrast, the following assignment statement causes an out-of-bounds exception because a particular element is referenced and does not exist:

```
points[1]{x=10, y=20};
```

You can add elements to the array, then use a single statement to set values in all elements:

```
points.resize(2);
points{x=1, y=1};
```

### **Related concepts**

“Bracket syntax for dynamic access” on page 61

“Overview of EGL properties” on page 64

“References to variables in EGL” on page 59

“Scoping rules and “this” in EGL” on page 57

### **Related reference**

“Arrays” on page 75

“Data initialization” on page 564

“openUI” on page 726

---

## Arrays

EGL supports the following kinds of arrays:

- “Dynamic arrays”
- “Structure-field arrays” on page 79

In either case, the maximum number of supported dimensions is seven.

### Dynamic arrays

When you declare an array of primitive variables or of records (type `BasicRecord`, `DLISegment`, or `SQLRecord`), the array has an identity independent of the elements in the array:

- A set of functions are specific to the array, allowing you to grow or shrink the number of elements at run time.
- The array-specific property **maxSize** indicates how many elements are valid in the array. The default value is unbounded; the number of elements is limited only by the requirements of the target environment.

You do not need to specify a number of elements in your declaration, but if you do, that number indicates the initial number of elements. You also can specify the initial number of elements by listing a series of array constants in the declaration, as is possible only with primitive variables, not with records.

The syntax for declaring a dynamic array is shown in the following examples:

```
// An array of 5 elements or less
myDataItem01 CHAR(30)[] { maxSize=5 };

// An array of 6 elements or less,
// with 4 elements initially
myDataItem02 myDataItemPart[4] { maxSize=6 };

// An array that has no elements
// but whose maximum size is the largest possible
myRecord myRecordPart[];

// A 3-element array whose elements
// are assigned the values 1, 3, and 5
position int[] = [1,3,5];
```

You can use a literal integer to initialize the number of elements, but neither a variable nor a constant is valid.

When you declare an array of arrays, an initial number of elements is valid in the leftmost-specified dimension and in each subsequent dimension until a dimension lacks an initial number. The following declarations are valid:

```
// Valid, with maxsize giving the maximum
// for the first dimension
myInt01 INT[3][];
myInt02 INT[4][2][] {maxsize = 12};
myInt03 INT[7][3][1];

// In the next example, array constants indicate
// that the outer array initially has 3 elements.
// The first element of the outer array is
// an array of two elements (with values 1 and 2).
// The second element of the outer array is
// an array of three elements (with values 3, 4,5).
```

```
// The third element of the outer array is
// an array of two elements (with values 6 and 7).
myInt04 INT[] [] = [[1,2],[3,4,5],[6,7]];
```

In the following example, the syntax is not valid because (for instance) the array `myInt04` is declared as an array of no elements, but each of those elements is assigned 3 elements:

```
// NOT valid
myInt04 INT[] [3];
myInt05 INT[5] [] [2];
```

An array specified as a program or function parameter cannot specify the number of elements.

When your code references an array or an array element, these rules apply:

- An element subscript can be any numeric expression that resolves to an integer, but the expression cannot include a function invocation.
- If your code refers to a dynamic array but does not specify subscripts, the reference is to the array as a whole.

An out-of-memory situation is treated as a catastrophic error and ends the program.

## Dynamic-array functions

A set of functions and read-only variables are available for each dynamic array. In the following example, the array is called *series*:

```
series.reSize(100);
```

The name of the array may include a set of brackets, each containing an integer. An example is as follows:

```
series INT[] [];

// resizes the second element
// of series, which is an array of arrays
series[2].reSize(100);
```

In the following sections, substitute the array name for *arrayName* and note that the name may be qualified by a package name, a library name, or both.

### appendAll():

```
arrayName.appendAll(appendArray Array in)
```

This function does as follows:

- Appends to the array that is referenced by *arrayName*, adding a copy of the array that is referenced by *appendArray*
- Increments the array size by the number of added elements
- Assigns an appropriate index value to each of the appended elements

The elements of *appendArray* must be of the same type as the elements of *arrayName*.

### appendElement()

```
arrayName.appendElement(content ArrayElement in)
```

This function places an element to the end of the specified array and increments the size by one. For *content*, you can substitute a variable of the appropriate type;

alternatively, you can specify a literal that is assigned to an element created during the operation. The process copies data; if you assign a variable, that variable is still available for comparison or other purposes.

The rules for assigning a literal are as specified in *Assignments*.

### **getMaxSize()**

`arrayName.getMaxSize ( )` returns (**INT**)

This function returns an integer that indicates the maximum of elements allowed in the array.

### **getSize()**

`arrayName.getSize ( )` returns (**INT**)

This function returns an integer that indicates the number of elements in the array. It is recommended that you use this function instead of *SysLib.size* when you are working with dynamic arrays.

Another function provides functionality equivalent to that of *arrayName.getSize*:

**SysLib.size( )** returns (**INT**)

However, it is recommended that you use *arrayName.getSize ( )* when you work with dynamic arrays.

### **insertElement()**

`arrayName.insertElement (content ArrayElement in, index INT in)`

This function does as follows:

- Places an element in front of the element that is now at the specified location in the array
- Increments the array size by one
- Increments the index of each element that resides after the inserted element

*content* is the new content (a constant or variable of the appropriate type for the array), and *index* is an integer literal or a numeric variable that indicates the location of the new element.

If *index* is one greater than the number of elements in the array, the function creates a new element at the end of the array and increments the array size by one.

### **removeAll()**

`arrayName.removeAll ( )`

This function removes the elements of the array from memory. The array can be used, but its size is zero.

### **removeElement()**

`arrayName.removeElement(index INT in)`

This function removes the element at the specified location, decrements the array size by one, and decrements the index of each element that resides after the removed element.



*index* is an integer literal or a numeric variable that indicates the location of the element to be removed.

### **resize()**

*arrayName.resize(size INT in)*

This function adds or shrinks the current size of the array to the size specified in *size*, which is an integer literal, constant, or variable. If the value of *size* is greater than the maximum size allowed for the array, the run unit terminates.

### **reSizeAll()**

*arrayName.reSizeAll(sizes INT[  
] in)*

This function adds or shrinks every dimension of a multidimensional array. The parameter *sizes* is an array of integers, with each successive element specifying the size of a successive dimension. If the number of dimensions resized is greater than the number of dimensions in *arrayName*, or if a value of an element in *sizes* is greater than the maximum size allowed in the equivalent dimension of *arrayName*, the run unit terminates.

### **setMaxSize()**

*arrayName.setMaxSize (size INT in)*

The function sets the maximum of elements allowed in the array. If you set the maximum size to a value less than the current size of the array, the run unit terminates.

### **setMaxSizes()**

*arrayName.setMaxSizes(sizes INT[  
] in)*

This function sets every dimension of a multidimensional array. The parameter *sizes* is an array of integers, with each successive element specifying the maximum size of a successive dimension. If the number of dimensions specified is greater than the number of dimensions in *arrayName*, or if a value of an element in *sizes* is less than the current number of elements in the equivalent dimension of *arrayName*, the run unit terminates.

## **Use of dynamic arrays as arguments and parameters**

A dynamic array can be passed as an argument to an EGL function. The related parameter must be defined as a dynamic array of the same type as the argument; and for a data item, the type must include the same length and decimal places, if any.

A dynamic array of the following types can be passed to a program:

- A primitive type
- A record of type BasicRecord, DLISegment, or SQLRecord

An example of a function that uses a dynamic array as a parameter is as follows:

```
Function getAll (employees Employee[])  
;  
end
```

At run time, the maximum size for a parameter is the maximum size declared for the corresponding argument. The invoked function can change the size of the array, and the change is in effect in the invoking code.

## SQL processing and dynamic arrays

EGL lets you use a dynamic array to access rows of a relational database. For details on reading multiple rows, see *get*. For details on adding multiple rows, see *add*.

## Structure-field arrays

You declare a structure-field array when you specify that a field in a fixed structure has an occurs value greater than one, as in the following example:

```
Record myFixedRecordPart
  10 mySi CHAR(1) [3];
end
```

If a fixed record called *myRecord* is based on that part, the symbol *myRecord.mySi* refers to a one-dimensional array of three elements, each a character.

## Usage of a structure-field array

You can reference an entire array of structure fields (for example, *myRecord.mySi*) in these contexts:

- As the second operand used by an *in* operator. The operator tests whether a given value is contained in the array.
- As the parameter in the function **sysLib.size**. That function returns the occurs value of the structure field.

An array element that is not itself an array is a field like any other, and you can reference that field in various ways; for example, in an assignment statement or as an argument in a function invocation.

An element subscript can be any numeric expression that resolves to an integer, but the expression cannot include a function invocation.

## One-dimensional structure-field array

You can refer to an element of a one-dimensional array like *myRecord.mySi* by using the name of the array followed by a bracketed subscript. The subscript is either an integer or a field that resolves to an integer; for example, you can refer to the second element of the example array as *myStruct.mySi[2]*. The subscript can vary from 1 to the occurs value of the structure field, and a runtime error occurs if the subscript is outside of that range.

If you use the name of a structure-field array in a context that requires a field but do not specify a bracketed subscript, EGL assumes that you are referring to the first element of the array, but only if you are in VisualAge Generator compatibility mode. It is recommended that you identify each element explicitly. If you are not in VisualAge Generator compatibility mode, you are required to identify each element explicitly.

The next examples show how to refer to elements in a one-dimensional array. In those examples, *valueOne* resolves to 1 and *valueTwo* resolves to 2:

```
// these refer to the first of three elements:
myRecord.mySi[valueOne]

// not recommended; and valid
// only if VisualAge Generator
```

```
// compatibility is in effect:
myRecord.mySi

// this refers to the second element:
myRecord.mySi[valueTwo]
```

A one-dimensional array may be substructured, as in this example:

```
record myRecord01Part
  10 name[3];
  20 firstOne CHAR(20);
  20 midOne CHAR(20);
  20 lastOne CHAR(20);
end
```

If a record called *myRecord01* is based on the previous part, the symbol *myRecord01.name* refers to a one-dimensional array of three elements, each of which has 60 characters, and the length of *myRecord01* is 180.

You may refer to each element in *myRecord01.name* without reference to the substructure; for example, *myRecord01.name[2]* refers to the second element. You also may refer to a substructure within an element. If uniqueness rules are satisfied, for example, you can reference the last 20 characters of the second element in any of the following ways:

```
myRecord01.name.lastOne[2]
myRecord01.lastOne[2]
lastOne[2]
```

The last two are valid only if the generatable-part property **allowUnqualifiedItemReferences** is set to *yes*.

For details on the different kinds of references, see *References to variables and constants*.

## Multidimensional structure-field array

If a structure item with an occurs value greater than one is substructured and if a subordinate structure item also has an occurs value greater than one, the subordinate structure item declares an array with an additional dimension.

Let's consider another record part:

```
record myRecord02Part
  10 siTop[3];
  20 siNext CHAR(20)[2];
end
```

If a record called *myRecord02* is based on that part, each element of the one-dimensional array *myRecord02.siTop* is itself a one-dimensional array. For example, you can refer to the second of the three subordinate one-dimensional arrays as *myRecord02.siTop[2]*. The structure item *siNext* declares a two-dimensional array, and you can refer to an element of that array by either of these syntaxes:

```
// row 1, column 2.
// the next syntax is strongly recommended
// because it works with dynamic arrays as well
myRecord02.siTop[1].siNext[2]

// the next syntax is supported for compatibility
// with VisualAge Generator
myRecord02.siTop.siNext[1,2]
```

To clarify what area of memory is being referenced, let's consider how data in a multidimensional array is stored. In the current example, `myRecord02` constitutes 120 bytes. The referenced area is divided into a one-dimensional array of three elements, each 40 bytes:

```
siTop[1]      siTop[2]      siTop[3]
```

Each element of the one-dimensional array is further subdivided into an array of two elements, each 20 bytes, in the same area of memory:

```
siNext[1,1] siNext[1,2] siNext[2,1] siNext[2,2] siNext[3,1] siNext[3,2]
```

A two-dimensional array is stored in row-major order. One implication is that if you initialize an array in a double while loop, you get better performance by processing the columns in one row before processing the columns in a second:

```
// i, j, myTop0ccurs, and myNext0ccurs are data items;
// myRecord02 is a record; and
// sysLib.size() returns the occurs value of a structure item.
i = 1;
j = 1;
myTop0ccurs = sysLib.size(myRecord02.siTop);
myNext0ccurs = sysLib.size(myRecord02.siTop.siNext);
while (i <= myTop0ccurs)
  while (j <= myNext0ccurs)
    myRecord02.siTop.siNext[i,j] = "abc";
    j = j + 1;
  end
  i = i + 1;
end
```

You must specify a value for each dimension of a multidimensional array. The reference `myRecord02.siTop.siNext[1]`, for example, is not valid for a 2-dimensional array.

An example declaration of a 3-dimensional array is as follows:

```
record myRecord03Part
  10 siTop[3];
  20 siNext[2];
  30 siLast CHAR(20)[5];
end
```

If a record called `myRecord03` is based on that part and if uniqueness rules are satisfied, you can reference the last element in the array in any of the following ways:

```
// each level is shown, and a subscript
// is on each level, as is recommended.
myRecord03.siTop[3].siNext[2].siLast[5]

// each level shown, and subscripts are on lower levels
myRecord03.siTop.siNext[3,2].siLast[5]
myRecord03.siTop.siNext[3][2].siLast[5]

// each level is shown, and subscripts are on the lowest level
myRecord03.siTop.siNext.siLast[3,2,5]
myRecord03.siTop.siNext.siLast[3,2][5]
myRecord03.siTop.siNext.siLast[3][2,5]
myRecord03.siTop.siNext.siLast[3][2][5]

// the container and the last level is shown, with subscripts
myRecord03.siLast[3,2,5]
myRecord03.siLast[3,2][5]
myRecord03.siLast[3][2,5]
myRecord03.siLast[3][2][5]
```

```
// only the last level is shown, with subscripts
siLast[3,2,5]
siLast[3,2][5]
siLast[3][2,5]
siLast[3][2][5]
```

As indicated by the previous example, you reference an element of a multidimensional array by adding a bracketed set of subscripts, in any of various ways. In all cases, the first subscript refers to the first dimension, the second subscript refers to the second dimension, and so forth. Each subscript can vary from 1 to the occurs value of the related structure item, and a runtime error occurs if a subscript resolves to a number outside of that range.

First, consider the situation when subscripts are not involved:

- You can specify a list that begins with the name of the variable and continues with the names of increasingly subordinate structure items, with each name separated from the next by a period, as in this example:

```
myRecord03.siTop.siNext.siLast
```

- You can specify the name of the variable, followed by a period, followed by the name of the lowest-level item of interest, as in this example:

```
myRecord03.siLast
```

- If the lowest-level item of interest is unique in a given name space, you can specify only that item, as in this example:

```
siLast
```

Next, consider the rules for placing array subscripts:

- You can specify a subscript at each level where one of several elements is valid, as in this example:

```
myRecord03.siTop[3].siNext[2].siLast[5]
```

- You can specify a series of subscripts at any level where one of several elements is valid, as in this example:

```
myRecord03.siTop.siNext[3,2].siLast[5]
```

- You can specify a series of subscripts at any level that is at or subordinate to a level where one of several elements is valid, as in this example:

```
myRecord03.siTop.siNext.siLast[3,2,5]
```

- An error occurs if you assign more subscripts than are appropriate at a given level. as in this example:

```
// NOT valid
myRecord03.siTop[3,2,5].siNext.siLast
```

- You can isolate a subscript within a bracket or can display a series of subscripts, each separated from the next by a comma; or you can combine the two usages.

The following examples are valid:

```
myRecord03.siTop.siNext.siLast[3,2,5]
myRecord03.siTop.siNext.siLast[3,2][5]
myRecord03.siTop.siNext.siLast[3][2,5]
myRecord03.siTop.siNext.siLast[3][2][5]
```

### Related concepts

“Compatibility with VisualAge Generator” on page 532

“References to variables in EGL” on page 59

### Related reference

“add” on page 661

“Assignments” on page 456  
“EGL system limits” on page 590  
“get” on page 687  
“in operator” on page 629  
“size()” on page 1040

---

## Dictionary

A *dictionary part* is a part that is always available; you do not define it. A variable that is based on a dictionary part may include a set of keys and their related values, and you can add and remove key-and-value entries at run time. The entries are treated like fields in a record.

An example of a dictionary declaration is as follows:

```
row Dictionary
{
    ID          = 5,
    lastName    = "Twain",
    firstName   = "Mark",
};
```

When you include entries in the declaration, each key name is an EGL identifier that must be consistent with the EGL naming conventions. When you add entries at run time, you have greater flexibility; you can specify a string literal, constant, or variable, and in this case the content can be an EGL reserved word or can include characters that would not be valid in an identifier. For details, see *Bracket syntax for dynamic access*.

Example assignments are as follows:

```
row.age = 30;
row["Credit"] = 700;
row["Initial rating"] = 500
```

If you attempt to assign a key that already exists, you override the existing key-and-value entry. The following assignment is valid and replaces "Twain" with "Clemens":

```
row.lastname = "Clemens";
```

Assignments also can be used for data retrieval:

```
lastname String
age, credit, firstCredit int;

lastname = row.lastname;
age = row.age;
credit = row.credit;
credit = row["Credit"];
firstCredit = row["Initial rating"];
```

The value in a key-and-value entry is of type ANY, which means that you can put different kinds of information into a single dictionary. Each value can be any of these:

- A pre-declared record or other variable
- A constant or literal

Putting a variable into a dictionary assigns a copy of the variable. Consider the following record part:

```

Record myRecordPart
  x int;
end

```

The next code places a variable of type `myRecordPart` into the dictionary, then changes a value in the original variable:

```

testValue int;

myRecord myRecordPart;

// sets a variable value and places
// a copy of the variable into the dictionary.
myRecord.x = 4;
row Dictionary
{
  theRecord myRecord;
}

// Places a new value in the original record.
myRecord.x = 700;

// Accesses the dictionary's copy of the record,
// assigning 4 to testValue.
testValue = row.theRecord.x;

```

Assigning one dictionary to another replaces the target content with the source content and overrides the values of the target dictionary's properties, which are described later. The conditional statement in the following code is true, for example:

```

row Dictionary { age = 30 };

newRow Dictionary { };
newRow = row

// resolves to true
if (newRow.age == 30)
;
end

```

A set of properties in the declaration affect how the dictionary is processed. A set of dictionary-specific functions provide data and services to your code.

## Dictionary properties

Each property-and-value entry is syntactically equivalent to a key-and-value entry, as shown in this example, and the entries can be in any order:

```

row Dictionary
{
  // properties
  caseSensitive = no,
  ordering = none,

  // fields
  ID = 5,
  lastName = "Twain",
  firstName = "Mark",
  age = 30;
};

```

Your code can neither add nor retrieve a property or its value. In the unlikely event that you wish to use a property name as a key, use the variable name as a qualifier when you specify or refer to the key, as in this example:

```

row Dictionary
{
    // properties
    caseSensitive = no,
    ordering = none,

    // fields
    row.caseSensitive = "yes"
    row.ordering = 50,
    age = 30
};

```

Properties are as follows:

### **caseSensitive**

Indicates whether retrieval of a key or the related value is affected by the case of the key with which that value was stored. Options are as follows:

#### **No (the default)**

Key access is unaffected by the case of the key, and the following statements are equivalent:

```

age = row.age;
age = row.AGE;
age = row["aGe"];

```

#### **Yes**

The following statements can have different results, even though EGL is primarily a case-insensitive language:

```

age = row.age;
age = row.AGE;
age = row["aGe"];

```

The value of the property **caseSensitive** affects the behavior of several of the functions described in a later section.

### **ordering**

Indicates how key-and-value entries are ordered for purpose of retrieval. The value of this property affects the behavior of the functions **getKeys** and **getValues**, as described in a later section.

Options are as follows:

#### **None (the default)**

Your code cannot rely on the order of key-and-value entries.

When the value of the property **ordering** is **None**, the ordering of keys (when the function **getKeys** is invoked) may not be the same as the ordering of values (when the function **getValues** is invoked).

#### **ByInsertion**

The key-and-value pairs are available in the order in which they were inserted. Any entries in the declaration are considered to be inserted first, in left-to-right order.

#### **ByKey**

The key-and-value pairs are available in key order.

## **Dictionary functions**

To invoke any of the following functions, qualify the function name with the name of the dictionary, as in this example when the dictionary is called *row*:



```

    if (row.containsKey(age))
    ;
end

```

## containsKey()

*dictionaryName.containsKey(key String in)* returns (Boolean)

This function resolves to true or false, depending on whether the input string (*key*) is a key in the dictionary. If the dictionary property **caseSensitive** is set to no, case is not considered; otherwise, the function seeks an exact match, including by case.

**containsKey** is used only in a logical expression.

## getKeys()

*dictionaryName.getKeys ( )* returns (String[ ])

This function returns an array of strings, each of which is a key in the dictionary.

If the dictionary property **caseSensitive** is set to no, each returned key is in lower case; otherwise, each returned key is in the case in which the key was stored.

If the dictionary property **ordering** is set to no, you cannot rely on the order of the returned keys; otherwise, the order is as specified in the description of that property.

## getValues()

*dictionaryName.getValues ( )* returns (ANY[ ])

This function returns an array of values of any type. Each value is associated with a key in the dictionary.

## insertAll()

*dictionaryName.insertAll(sourceDictionary Dictionary in)*

This function acts as if a series of assignment statements copies the key-and-value entries in the source dictionary (*sourceDictionary*) to the *target*, which is the dictionary whose name qualifies the function name.

If a key is in the source and not in the target, the key-and-value entry is copied to the target. If a key is in both the source and target, the value of the source entry overrides the entry in the target. The determination of whether a key is in the target matches one in the source is affected by the value of property **caseSensitive** in each dictionary.

This function is different from the assignment of one dictionary to another because the function **insertAll** retains these entries:

- The property-and-value entries in the target; and
- The key-and-value entries that are in the target but not the source.

## removeElement()

*dictionaryName.removeElement(key String in)*

This function removes the entry whose input string (*key*) is a key in the dictionary. If the dictionary property **caseSensitive** is set to no, case is not considered; otherwise, the function seeks an exact match, including by case.

## removeAll()

*dictionaryName.removeAll( )*

This function removes all key-and-value entries in the dictionary, but has no effect on the dictionary's properties.

## size()

*dictionaryName.size( )* returns (INT)

Returns an integer that indicates the number of key-and-value entries in the dictionary.

### Related concepts

"Parts" on page 19

"References to variables in EGL" on page 59

### Related reference

"Logical expressions" on page 593

---

## ArrayDictionary

An *arrayDictionary part* is a part that is always available; you do not define it. A variable that is based on an *arrayDictionary part* lets you access a series of arrays by retrieving the same-numbered element of every array. A set of elements that is retrieved in this way is itself a dictionary, with each of the original array names treated as a key that is paired with the value contained in the array element.

An *arrayDictionary* is especially useful in relation to the display technology described in *Console user interface*.

The next graphic illustrates an *arrayDictionary* whose declaration includes arrays that are named *ID*, *lastname*, *firstname*, and *age*. The ellipse encloses a dictionary that includes the following key-and-value entries:

```
ID = 5,  
lastName = "Twain",  
firstName = "Mark",  
age = 30
```

ID	LastName	FirstName	Age
5	Twain	Mark	30

The array of interest is the array of dictionaries, with each dictionary illustrated as being one above the next rather than one alongside the next. The declaration of the `arrayDictionary` requires an initial list of arrays, however, and those are illustrated as being one alongside the next.

The following code shows the declaration of a list of arrays, followed by the declaration of an `arrayDictionary` that uses those arrays:

```
ID          INT[4];
lastName    STRING[4];
firstName   STRING[4];
age         INT[4];

myRows ArrayDictionary
{
    col1 = ID,
    col2 = lastName,
    col3 = firstName,
    col4 = age
};
```

To retrieve values, your code uses a syntax that isolates a particular dictionary and then a particular field (a key-and-value entry) in that dictionary. You cannot use the `arrayDictionary` syntax to update a value or to change any characteristic of the `arrayDictionary` itself.

First, declare a dictionary and assign an `arrayDictionary` row to that dictionary, as in this example:

```
row Dictionary = myRows[2];
```

Next, declare a variable of the appropriate type and assign an element to that variable, as in either of these examples:

```
cell INT = row["ID"];

cell INT = row.ID;
```

An alternative syntax retrieves the value in one step, as in either of these examples:

```
cell int = myRows[2]["ID"];

cell int = myRows[2].ID;
```

### Related concepts

“Console user interface” on page 207

“Dictionary” on page 83

“References to variables in EGL” on page 59

---

## EGL statements

Each EGL function is composed of zero to many EGL statements of the following kinds:

- A *variable declaration* or *constant declaration* provides access to a named area of memory. The value of a variable can be changed at run time; the value of a constant cannot. Either kind of declaration can be anywhere in a function except in a *block*, as described later.
- A *function invocation* directs processing to a function, as in this example:

```
myFunction(myInput);
```

Recursive calls are valid.
- An *assignment statement* can copy any of the following values into a variable:

- Data from a constant or variable
- A literal
- A value returned from a function invocation
- The result of an arithmetic calculation
- The result of a string concatenation

Examples of assignment statements are as follows:

```
myItem = 15;
myItem = readFile(myKeyValue);
myItem = bigValue - 32;
record1.message = "Operation " + "successful!";
```

- A *keyword statement* provides additional functionality such as file access. Each of these statements is named for the keyword that begins the statement; for example:

```
add record1;    // an add statement
return (0);    // a return statement
```

- A *null statement* is a semicolon that has no effect but may be useful as a placeholder, as in this example:

```
if (myItem == 5)
;           // a null statement
else
  myFunction(myItem);
end
```

Non-null EGL statements have the following characteristics:

- A statement can reference named memory areas, which are of these kinds:
  - Form
  - PageHandler
  - Record
  - DataTable
  - Item (a category that includes data items, as well as structure items in records, forms, and tables)
  - Array (a memory area based on a structure item that has an occurs value greater than 1)
- A statement can include these kinds of expressions--
  - A *datetime expression* resolves to a date, integer, interval, or timestamp
  - A *logical expression* resolves to true or false
  - A *numeric expression* resolves to a number, which may be signed and include a decimal point
  - A *string expression* resolves to a series of characters, which may include single-byte characters, double-byte characters, or a combination of the two
- A statement either ends with a semicolon or with a *block*, which is a series of zero or more subordinate statements that act as a unit. Block-containing statements are terminated with an end delimiter, as in this example:

```
if (record2.status= "Y")
  record1.total = record1.total + 1;
  record1.message = "Operation successful!";
else
  record1.message = "Operation failed!";
end
```

A semicolon after an end delimiter is not an error but is treated as a null statement.

Names in statements and throughout an EGL source file are case-*insensitive*; *record1* is identical to *RECORD1*, for example, and both *add* and *ADD* refer to the same keyword. Exceptions are mentioned as appropriate, as in the following cases--

- The name of a generatable part such as a program must be the same as the name of file, and the name is case sensitive.
- The name of an EGL package is case sensitive.
- When you use the source tab in Page Designer, you can manually bind components in a JSP file (specifically, in a JavaServer Faces file) to data areas in a PageHandler. Although EGL is not case sensitive, EGL variable names referenced in the JSP file must have the same case as the EGL variable declaration; and you fail to maintain an exact match, a JavaServer Faces error occurs. It is recommended that you avoid changing the case of an EGL variable after you bind that variable to a JSP field.

*System words* are a set of words that provide special functionality:

- A *system function* runs code and may return a value; for example:
  - **sysLib.minimum(arg1, arg2)** returns the minimum of two numbers
  - **strLib.strLen(arg1)** returns the length of a character stringThe qualifier (**mathLibstrLib** or **sysLib**) is necessary only if your program has a function of the same name.
- A *system variable* provides a value without invoking a function; for example:
  - **sysVar.errorCode** contains a status code after your program accesses a file and in other situations
  - **sysVar.sqlcode** contains a status code after your program accesses a relational databaseThe qualifier **sysVar** is necessary only if your program has a variable of the same name.

A line in a function can have more than one statement. It is recommended that you include no more than one statement per line, however, because you can use the EGL Debugger to set a breakpoint only at the first statement on a line.

See also *Comments*.

### Related concepts

“EGL projects, packages, and files” on page 15

“Function part” on page 150

“Parts” on page 19

### Related reference

“Assignments” on page 456

“Comments” on page 531

“Data initialization” on page 564

“EGL reserved words” on page 581

“Function invocations” on page 613

“Keyword statements in alphabetical order” on page 91

“Logical expressions” on page 593

“Numeric expressions” on page 600

“Text expressions” on page 601

“terminalID” on page 1075

## Keyword statements in alphabetical order

Keyword	Purpose
"add" on page 661	Places a record in a file, message queue, or database; or places a set of records in a database.
"call" on page 665	Transfers control to another program and optionally passes a series of values. Control returns to the caller when the called program ends. If the called program changes any data that was passed by way of a variable, the storage area available to the caller is changed, too.
"case" on page 668	Marks the start of multiple sets of statements, where at most only one of those sets is run. The <b>case</b> statement is equivalent to a C or Java <i>switch</i> statement that has a break at the end of each case clause.
"close" on page 669	Disconnects a printer; or closes the file or message queue associated with a given record; or, in the case of an SQL record, closes the cursor that was opened by an EGL <b>open</b> or <b>get</b> statement.
"continue" on page 672	Transfers control to the end of a <b>for</b> , <b>forEach</b> , <b>openUI</b> , or <b>while</b> statement that itself contains the <b>continue</b> statement.
"converse" on page 672	Presents a text form in a text application or presents a VGUI record in a Web application.
"delete" on page 673	Removes either a record from a file or a row from a database.
"display" on page 676	Adds a text form to a runtime buffer but does not present data to the screen; but is available only in VisualAge Generator compatibility mode.
"execute" on page 677	Lets you write one or more SQL statements; in particular, SQL data-definition statements (of type CREATE TABLE, for example) and data-manipulation statements (of type INSERT or UPDATE, for example).
"exit" on page 681	Leaves the specified block, which by default is the block that immediately contains the <b>exit</b> statement.
"for" on page 683	Begins a statement block that runs in a loop for as many times as a test evaluates to true.
"forEach" on page 684	Marks the start of a set of statements that run in a loop. The first iteration occurs only if a specified result set is available and continues (in most cases) until the last row in that result set is processed.
"forward" on page 686	Used primarily to display a Web page with variable information; but also can access a URL or can invoke a servlet or Java program that runs in the Web application server.
"freeSQL" on page 687	Frees any resources associated with a dynamically prepared SQL statement, closing any open cursor associated with that SQL statement.
"get" on page 687	Retrieves a single file record or database row and provides an option that lets you replace or delete the stored data later in your code. In addition, this statement allows you to retrieve a set of database rows and place each succeeding row into the next SQL record in a dynamic array. The <b>get</b> statement is sometimes identified as <b>get by key value</b> and is distinct from the <b>get by position</b> statements like <b>get next</b> .
"get absolute" on page 695	Reads a numerically specified row in a relational-database result set that was selected by an <b>open</b> statement.
"get current" on page 697	Reads the row at which the cursor is already positioned in a database result set that was selected by an <b>open</b> statement.
"get first" on page 698	Reads the first row in a database result set that was selected by an <b>open</b> statement.

Keyword	Purpose
"get last" on page 699	Reads the last row in a database result set that was selected by an <b>open</b> statement.
"get next" on page 701	Reads the next record from a file or message queue, or the next row in a database result set.
"get" on page 687	As used for DL/I access, reads the next child segment that has the same parent as the segment at the current database position.
"get previous" on page 708	Reads the previous record in the file that is associated with a specified EGL indexed record; or reads the previous row in a database result set that was selected by an <b>open</b> statement.
"get relative" on page 713	Reads a numerically specified row in a database result set that was selected by an <b>open</b> statement. The row is identified in relation to the cursor position in the result set.
"goTo" on page 714	Causes processing to continue at a specified label, which must be in the same function as the statement and outside of a block.
"if, else" on page 715	Marks the start of a set of statements (if any) that run only if a logical expression resolves to true. The optional keyword <b>else</b> marks the start of an alternative set of statements (if any) that run only if the logical expression resolves to false. The reserved word <b>end</b> marks the close of the <b>if</b> statement.
"move" on page 716	Copies data, either byte by byte or by name. The latter operation copies data from the named items in one structure to the same-named items in another.
"open" on page 722	Selects a set of rows from a relational database for later retrieval with <b>get by position</b> statements like <b>get next</b> . The <b>open</b> statement may operate on a cursor or on a called procedure.
"openUI" on page 726	Allows the user to interact with a program whose interface is based on consoleUI. The <b>openUI</b> statement defines user and program events and specifies how to respond to each.
"prepare" on page 736	Specifies an SQL PREPARE statement, which optionally includes details that are known only at run time. You run the prepared SQL statement by running an EGL <b>execute</b> statement or (if the SQL statement returns a result set) by running an EGL <b>open</b> or <b>get</b> statement.
"print" on page 737	Adds a print form to a runtime buffer.
"replace" on page 738	Puts a changed record into a file or database.
"return" on page 741	Exits from a function and optionally returns a value to the invoking function.
"set" on page 742	Has various effects on records, text forms, and fields.
"show" on page 751	Presents a text form from a main program along with any other forms buffered using the display statement; ends the current program and optionally forwards the input data from the user and state data from the current program to the program that handles the input from the user.
"transfer" on page 752	Gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's input record. You cannot use a <b>transfer</b> statement in a called program.
"try" on page 754	Indicates that the program continues running if an input/output (I/O) statement, a system-function invocation, or a <b>call</b> statement results in an error and is within the <b>try</b> statement.

Keyword	Purpose
"while" on page 755	Marks the start of a set of statements that run in a loop. The first run occurs only if a logical expression resolves to true, and each subsequent iteration depends on the same test. The reserved word <b>end</b> marks the close of the <b>while</b> statement.

#### Related reference

"EGL statements" on page 88

## Transfer of control across programs

EGL provides several ways to switch control from one program to another:

- The **call** statement gives control to another program and optionally passes a series of values. Control returns to the caller when the called program ends. If the called program changes any data that was passed as a variable, the content of the variable is changed in the caller.

The call does not commit databases or other recoverable resources, although an automatic server-side commit may occur.

You may specify characteristics of the call by setting a `callLink` element of the linkage options part. For details, see *call* and *callLink element*. For details on the automatic server-side commit, see *luwControl* in *callLink element*.

- The **transfer** statement gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's *input record*. You cannot use a **transfer** statement in a called program.

Your program can transfer control by a statement of the form *transfer to a transaction* or by a statement of the form *transfer to a program*:

- A transfer to a transaction acts as follows--
  - In a program that runs as a Java main text or main batch program, the behavior depends on the setting of build descriptor option `synchOnTrxTransfer`--
    - If the value of `synchOnTrxTransfer` is YES, the transfer statement commits recoverable resources, closes files, closes cursors, and starts a program in the same run unit.
    - If the value of `synchOnTrxTransfer` is NO (the default), the transfer statement also starts a program in the same run unit, but does not close or commit resources, which are available to the invoked program.
  - A *transfer to a program* does not commit or rollback recoverable resources, but closes files, releases locks, and starts a program in the same run unit.

The linkage options part does not affect the characteristics of either kind of transfer.

In a `PageHandler`, a transfer is not valid.

For details, see *transfer*.

- The system function **sysLib.startTransaction** starts a run unit asynchronously. The operation does not end the transferring program and does not affect the databases, files, and locks in the transferring program. You have the option to pass data into the *input record*, which is an area in the receiving program.

If your program invokes **sysLib.startTransaction**, you must generate the program with a linkage options part, **asynchLink** element. For details, see *sysLib.startTransaction* and *asynchLink element*.



- The EGL **show** statement ends a main textUI program or a main VGWebTransaction program and shows data to the user. After the user submits the form or Web page, the **show** statement optionally forwards control to a second main program, which receives data received from the user as well as data that was passed without change from the originating program.

In relation to text applications, the **show** statement is affected by the settings in the linkage options part, **transferLink** element.

For details, see *show*.

- Finally, the **forward** statement is invoked from a PageHandler or program. The statement acts as follows:

1. Commits recoverable resources, closes files, and releases locks
2. Forwards control
3. Ends the code

The target in this case is another program or a Web page. For details, see *forward*.

For details that are specific to IMS, see *Transfers to and from EGL-generated MPPs*,

#### Related reference

"Transfers to and from EGL-generated IMS MPPs" on page 627

#### Related reference

"asynchLink element" on page 459

"call" on page 665

"callLink element" on page 499

"forward" on page 686

"luwControl in callLink element" on page 506

"show" on page 751

"startTransaction()" on page 1041

"transfer" on page 752

---

## Exception handling

An error may occur when an EGL-generated program acts as follows:

- Accesses a file, queue, or database
- Calls another program
- Invokes a function
- Performs an assignment, comparison, or calculation

### try blocks

An EGL *try block* is a series of zero to many EGL statements within the delimiters **try** and **end**. An example is as follows:

```
if (userRequest = "A")
  try
    add record1;
  onException
    myErrorHandler(12);
end
end
```

In general, a try block allows your program to continue processing even if an error occurs.

The try block may include an *onException clause*, as shown earlier. That clause is invoked if one of the earlier statements in the try block fails; but in the absence of an onException clause, an error in a try block causes invocation of the first statement that immediately follows the try block.

## System exceptions

EGL provides a series of system exceptions to indicate the specific nature of a runtime problem. Each of these exceptions is a dictionary from which you can retrieve information, but your retrieval is always by way of the system variable **SysLib.currentException** (also a dictionary), which lets you access the exception thrown most recently in the run unit.

One field in any exception is **code**, which is a string that identifies the exception. You can determine the current exception by testing that field in logic like this:

```
if (userRequest = "A")
  try
    add record1;
  onException
    case (SysLib.currentException.code)
      when (FileIOException)
        myErrorHandler(12);
      otherwise
        myErrorHandler(15);
    end
  end
end
```

In this case, FileIOException is a constant, which is equivalent to the string value "com.ibm.egl.FileIOException". The EGL exception constant is always equivalent to the last qualifier in a string that begins "com.ibm.egl".

It is strongly recommended that you access the exception fields only in an onException block. The run unit terminates if your code accesses **SysLib.currentException** when no exception has occurred.

The next example accesses the field sqlcode in the exception SQLException:

```
if (userRequest = "A")
  try
    add record01;
  onException
    case (SysLib.currentException.code)
      when ("com.ibm.egl.SQLException")
        if (SysLib.currentException.sqlcode == -270)
          myErrorHandler(16);
        else
          myErrorHandler(20);
        end
      otherwise
        myErrorHandler(15);
    end
  end
end
```

For details on the system exceptions, see *EGL system exceptions*.

## Limits of try blocks

The previous details on try blocks must be qualified. First, a try block affects processing only for errors in the following kinds of EGL statements:

- An I/O statement

- A system function
- A call statement

Processing of numeric overflows is not affected by the presence of a try block. For details on those kinds of error, see *VGVar.handleOverflow*.

Second, a try block has no effect on errors inside a user function (or program) that is invoked from within the try block. In the next example, if a statement fails in function `myABC`, the program ends immediately with an error message unless function `myABC` itself handles the error:

```
if (userRequest = "B")
  try
    myVariable = myABC();
  onException
    myErrorHandler(12);
end
end
```

Third, the program ends immediately and with an error message in the following cases:

- An error of a kind that is covered specifically by a try block occurs outside of a try block; or
- One of the following cases applies, even in a try block--
  - A user-written function fails at invocation or on return to the invoker; or
  - The system variable **VGVar.handleHardIOErrors** is set to zero when a file or MQSeries I/O statement ends with a hard error (as described later); or
  - The system variable **DLIVar.handleHardDLIErrors** and **VGVar.handleHardIOErrors** are both set to zero when an attempt to access an IMS queue or DL/I database ends with a hard error.

The following cases are also of interest:

- If a value is divided by zero, a Java program handles the situation as a numeric overflow
- A Java program ends if a non-numeric character is assigned to a numeric variable

**Note:** To support the migration of programs written in VisualAge Generator and EGL 5.0, the variable **VGVar.handleSysLibraryErrors** (previously called `ezereply`) allows you to process some errors that occur outside of a try block. Avoid use of that variable, which is available only if you are working in VisualAge Generator compatibility mode.

## Error-related system variables

EGL provides error-related system variables that are set in a try block either in response to successful events or in response to non-terminating errors. The values in those variables are available in the try block and in code that runs subsequent to the try block, and the values in most cases are restored after a converse, if any.

Within a try block, the following rules apply:

- The system variable **sysVar.errorCode** is given a value after any of the following kinds of statements run:
  - A **call** statement
  - An I/O statement that operates on an indexed, MQ, relative, or serial file

- An invocation of almost any system function
  - After an I/O statement operates on an MQ record, the system variables **sysVar.errorCode** and **VGVar.mqConditionCode** are given values
  - After a relational database is accessed in a try block, values are assigned to variables in the SQL communication area (SQLCA); for details, see *SysVar.sqlca*
  - After a DL/I database is accessed, values are assigned to the PCB status variables (**DLIVar.dbName**, **DLIVar.keyArea**, **DLIVar.keyAreaLen**, **DLIVar.segmentLevel**, **DLIVar.procOptions**, **DLIVar.segmentName**, **DLIVar.numSensitiveSegs**, and **DLIVar.statusCode**).
- If you are running in CICS, values are also assigned to the CICS DL/I status variables (**DLIVar.cicsError**, **DLIVar.cicsCondition**, and **DLIVar.cicsRestart**).

EGL runtime does not set the DL/I-related variables listed earlier after your code accesses a GSAM file or an IMS message queue access. That access is considered to be serial-file access. To gain access to additional status information in those cases, use the appropriate PCB record. For an overview, see *EGL support for runtime PSBs and PCBs*. For details on particular types of PCB records, see *Record types and properties*.

An error code is assigned to **sysVar.errorCode** in the case of a non-terminating numeric overflow, as described in *VGVar.handleOverflow*; but a successful arithmetic calculation does not affect any error-related system variable.

Error-related system variables are also not affected by the invocation of a function other than a system function, and **sysVar.errorCode** (the variable affected by most system functions) is not affected by errors in these:

- **sysLib.calculateChkDigitMod10**
- **sysLib.calculateChkDigitMod11**
- **strLib.concatenate**
- **strLib.concatenateWithSeparator**
- **VGLib.concatenateBytes**
- **VGLib.connectionService**
- **sysLib.connect**
- **sysLib.convert**
- **sysLib.disconnect**
- **sysLib.disconnectAll**
- **sysLib.purge**
- **sysLib.queryCurrentDatabase**
- **strLib.setBlankTerminator**
- **sysLib.setCurrentDatabase**
- **strLib.strLen**
- **sysLib.verifyChkDigitMod10**
- **sysLib.verifyChkDigitMod11**
- **sysLib.wait**

The EGL runtime does not change the value of any error-related variables when statements run outside of a try block. Your program, however, may assign a value to an error-related variable outside of a try block.

## I/O statements

In relation to I/O statements, an error can be hard or soft:

- A soft error is any of these--
  - No record was found during an I/O operation on an SQL database table

- One of the following problems occurs in an I/O operation on an indexed, relative, or serial file; on a DL/I segment; or in other DL/I access:
  - Duplicate record (when the external data store allows insertion of a duplicate). The equivalent DL/I status code is II.
  - No record found. The equivalent DL/I status code is GE, and the equivalent IMS message-queue status code is QD.
  - End of file. The equivalent DL/I or GSAM status code is GB, and the equivalent IMS message-queue status code is QC,
- In most cases, a hard error is any other problem; for example--
  - Duplicate record (when the external data store prohibits insertion of a duplicate)
  - File not found
  - Communication links are not available during remote access of a data set

In relation to DL/I, a hard error is any non-blank status code other than GA, GB, GD, GE, GK, or II. In relation to GSAM, a hard error is any non-blank status code other than GB. In relation to IMS, a hard error is any non-blank status code other than QC, QD, CE, CF, CG, CI, CJ, CK, or CL.

If the statement that causes the soft error is in a try block, the following statements apply:

- By default, EGL continues running without passing control to the `onException` block
- If you wish to pass control to the `onException` block, set the property **`throwNrfEofExceptions`** to *yes* in a program, `pageHandler`, or library

If a hard I/O error occurs in a try block, the consequence depends on the value of an error-related system variable:

- During access of a file, relational database, or MQSeries message queue, the following rules apply--
  - If **`sysVar.handleHardIOErrors`** is set to 1, the program continues running
  - If **`sysVar.handleHardIOErrors`** is set to 0, the program presents an error message, if possible, and ends

The default setting of that variable is dependent on the value of the property **`handleHardIOErrors`**, which is available in generatable logic parts like programs, libraries, and `pageHandlers`. The default value for the property is *yes*, which sets the initial value of the variable **`sysVar.handleHardIOErrors`** to 1.

- During access of an IMS queue or DL/I database, the following rules apply--
  - If **`VGVar.handleHardIOErrors`** or **`DLIVar.handleHardDLIErrors`** is set to 1, the program continues running
  - If **`VGVar.handleHardIOErrors`** or **`DLIVar.handleHardDLIErrors`** is set to 0, the program presents an error message, if possible, and ends

If either a hard or soft I/O error occurs outside of a try block, the generated program presents an error message, if possible, and ends.

If you are accessing DB2 directly (not through JDBC), the `sqlcode` for a hard error is 304, 802, or less than 0.

## Error identification

You can determine what kind of error occurred in a try block by including a **`case`** or **`if`** statement inside or outside the try block, and in that statement you can test

the value of various system variables. If you are responding to an I/O error and if your statement uses an EGL record, however, it is recommended that you use an elementary logical expression. Two formats of the expression are available:

*recordName* **is** *IOerrorValue*

*recordName* **not** *IOerrorValue*

*recordName*

Name of the record used in the I/O operation

*IOerrorValue*

One of several I/O error values that are constant across database management systems

If you don't use the logical expressions with I/O error values and then change database management systems, you may need to modify and regenerate your program. In particular, it is recommended that you use the I/O error values to test for errors rather than the value of **sysVar.sqlcode** or **sysVar.sqlState**. Those values are dependent on the underlying database implementation.

#### **Related concepts**

"Compatibility with VisualAge Generator" on page 532

"Dictionary" on page 83

"EGL support for runtime PSBs and PCBs" on page 346

"Record types and properties" on page 138

#### **Related reference**

"EGL Java runtime error codes" on page 1097

"EGL statements" on page 88

"EGL system exceptions" on page 587

"I/O error values" on page 638

"Logical expressions" on page 593

"errorCode" on page 1068

"overflowIndicator" on page 1069

"sqlca" on page 1071

"sqlcode" on page 1072

"sqlState" on page 1073

"handleSysLibraryErrors" on page 1084

"handleHardIOErrors" on page 1082

"handleOverflow" on page 1083

"mqConditionCode" on page 1084

---

## **Event handling in EGL**

In programming, events are occurrences that you can clearly define. An event may be as simple as a mouse click or as complex as the initialization of variables before a report program begins to generate a new user-defined group. Events may reflect major milestones in a project, or they may be any of the countless tiny actions leading to that point.

An event handler, then, is a section of code that provide a response to a specific occurrence. If you have ever clicked a link in a Web browser, you have seen an event handler in action. The event was the click of your mouse button; an event handler then told the browser to load a specified page and possibly to start a new browser window.

EGL uses event handlers in three places:

- To provide complex functions when you print reports through the JasperReports API
- To create a PageHandler to control a user's runtime interaction with a Web page
- As part of an OnEventBlock in an openUI statement in Console UI.

**Related tasks**

"Migrating EGL code to EGL V6.0 iFix" on page 103

**Related reference**

"Predefined EGL report handler functions" on page 271

---

## Migrating EGL code to EGL V6.0.1

If your EGL code is only at the level of V5.1.2, use the migration tool described in *Migrating EGL code to EGL V6.0 iFix*.

To ready your V6.0 iFix code for EGL V6.0.1, make sure that none of your identifiers (such as variable or part names) begins with the at sign (@), which is now used as the first character of a complex property. Also, if you migrated code from VisualAge Generator, you may need to change **StrLib.compareStr** to **VGLib.compareBytes**, **StrLib.copyStr** to **VGLib.copyBytes**, and **StrLib.concatenate** to **VGLib.concatenateBytes**; but a change is needed only when the last argument in the existing invocation is a numeric value, as evidenced by an error message that indicates a problem with the argument's primitive type.

### Related tasks

"Migrating EGL code to EGL V6.0 iFix" on page 103

### Related concepts

"Setting EGL-to-EGL migration preferences" on page 112

"Changes to properties during EGL-to-EGL migration" on page 107

### Related reference

"EGL reserved words" on page 581





---

## Migrating EGL code to EGL V6.0 iFix

The EGL V6.0 migration tool converts EGL source from V5.1.2 and V6.0 to comply with the EGL V6.0 iFix. Additional changes for EGL V6.0.1 are described at the end of this topic.

**Note: Do not use the migration tool on code that has already been updated to the EGL V6.0 iFix. Doing so can create errors in your code.**

The migration tool can be used on an entire project, a single file, or a selection of files. Running the tool on a package or folder converts all of the EGL source files in that package or folder. For more information on the code that is changed by the migration tool, see *EGL-to-EGL migration*.

To migrate EGL code to the EGL V6.0 iFix, do as follows:

1. In the workbench, click **Window > Preferences**.
2. On the left side of the Preferences window, expand **Workbench** and click **Capabilities**.
3. From the list of capabilities, expand **EGL Developer**.
4. Select the check box for the capability named **EGL V6.0 Migration**.
5. Click **OK**.
6. Again, click **Window > Preferences**.
7. On the left side of the Preferences window, expand **EGL** and click **EGL V6.0 Migration Preferences**.
8. Set the preferences for the EGL V6.0 migration tool. For more information on the preferences in this window, see *Setting EGL-to-EGL migration preferences*.
9. In the Project Explorer view or the Navigator view, select the EGL projects, packages, folders, or files you want to migrate. You can select any number of EGL resources to migrate. To select more than one resource at once, hold CTRL while clicking the resources.
10. Right-click on a selected resource and click **EGL V6.0 Migration > Migrate** from the popup menu.
11. Inspect your code for places that do not comply with the EGL V6.0 iFix.

The migration tool converts the selected EGL source files to comply with the EGL V6.0 iFix. To review the changes that the tool made to the source code, do as follows:

1. In the Project Explorer view or the Navigator view, right-click an EGL source file that has been migrated and click **Compare With > Local History** from the popup menu.
2. Examine the differences between the file in the workspace and the previous version.
3. When you are finished reviewing the changes, click **OK**.

After you use the EGL V6.0 migration tool, make sure that none of your identifiers (such as variable or part names) begins with the at sign (@), which is now used as the first character of a complex property. Also, if you migrated code from VisualAge Generator, you may need to change **StrLib.compareStr** to **VGLib.compareBytes**, **StrLib.copyStr** to **VGLib.copyBytes**, and **StrLib.concatenate**

to `VGLib.concatenateBytes`; but a change is needed only when the last argument in the existing invocation is a numeric value, as evidenced by an error message that indicates a problem with the argument's primitive type.

#### Related concepts

"EGL-to-EGL migration"

"Setting EGL-to-EGL migration preferences" on page 112

#### Related tasks

"Enabling EGL capabilities" on page 124

---

## EGL-to-EGL migration

The EGL V6.0 migration tool converts EGL source from V5.1.2 and V6.0 to comply with the EGL V6.0 iFix. Additional changes for EGL V6.0.1 are described in full at the end of this topic.

**Note: Do not use the migration tool on code that has already been updated to the EGL V6.0 iFix. Doing so can create errors in your code.**

The migration tool can be used on an entire project, a single file, or a selection of files. Running the tool on a package or folder converts all of the EGL source files in that package or folder. For instructions on how to use the migration tool, see *Migrating EGL code to EGL 6.0.1*.

The migration tool can add comments to each file it changes, and it can also add comments to the project's log file. To change these options, see *EGL-to-EGL migration preferences*.

The migration tool changes EGL code in these ways to comply with the EGL V6.0 iFix:

- The migration tool makes changes to the way properties are specified. For information about the changes to properties, see *Changes to properties during EGL-to-EGL migration*.
- The migration tool searches for variables and part names that conflict with reserved words. The migration tool changes those variable and part names by adding a prefix or suffix as defined in the EGL-to-EGL migration preferences. By default, the tool adds the suffix `_EGL` to any name that is now a reserved word. The migration tool does not rename objects of the CALL statement, and it does not update references in EGL Build Part files. See *EGL reserved words*. The following is an example of code before and after using the migration tool.

Before migration:

```
Library Handler
  boolean Bin(4);
End
```

After migration:

```
Library Handler_EGL
  boolean_EGL Bin(4);
End
```

- The migration tool replaces the single equals sign (=) with the double equals sign (==) when used as a comparison operator. It does not change the single equals sign when used as an assignment operator.

Before migration:

```
Function test(param int)
  a int;
  If(param = 3)
    a = 0;
  End
End
```

After migration:

```
Function test(param int)
  a int;
  If(param == 3)
    a = 0;
  End
End
```

- The migration tool adds level numbers to records that do not have level numbers.

Before migration:

```
Record MyRecord
  item1 int;
  item2 int;
End
```

After migration:

```
Record MyRecord
  10 item1 int;
  10 item2 int;
End
```

- The migration tool changes the declaration syntax of constants.

Before migration:

```
intConst 3;
```

After migration:

```
const intConst int = 3;
```

- The migration tool changes variables and function names that have been moved to different libraries or renamed. This change affects variables and functions from the SysLib and SysVar libraries.

Before migration:

```
SysLib.java();
clearRequestAttr();
```

After migration:

```
JavaLib.invoke();
J2EELib.clearRequestAttr();
```

Following is a list of changed variables and function names from the SysLib and SysVar libraries:

*Table 1. Changed variable and function names from the SysLib and SysVar libraries*

Before migration	After migration
SysLib.dateValue	DateTimeLib.dateValue
SysLib.extendTimestampValue	DateTimeLib.extend
SysLib.formatDate	StrLib.formatDate
SysLib.formatTime	StrLib.formatTime
SysLib.formatTimestamp	StrLib.formatTimestamp
SysLib.intervalValue	DateTimeLib.intervalValue
SysLib.timeValue	DateTimeLib.timeValue
SysLib.timestampValue	DateTimeLib.timestampValue

*Table 1. Changed variable and function names from the SysLib and SysVar libraries (continued)*

<b>Before migration</b>	<b>After migration</b>
SysLib.java	JavaLib.invoke
SysLib.javaGetField	JavaLib.getField
SysLib.javaIsNull	JavaLib.isNull
SysLib.javaIsObjID	JavaLib.isObjID
SysLib.javaRemove	JavaLib.remove
SysLib.javaRemoveAll	JavaLib.removeAll
SysLib.javaSetField	JavaLib.setField
SysLib.javaStore	JavaLib.store
SysLib.javaStoreCopy	JavaLib.storeCopy
SysLib.javaStoreField	JavaLib.storeField
SysLib.javaStoreNew	JavaLib.storeNew
SysLib.javaType	JavaLib.qualifiedTypeName
SysLib.clearRequestAttr	J2EELib.clearRequestAttr
SysLib.clearSessionAttr	J2EELib.clearSessionAttr
SysLib.getRequestAttr	J2EELib.getRequestAttr
SysLib.getSessionAttr	J2EELib.getSessionAttr
SysLib.setRequestAttr	J2EELib.setRequestAttr
SysLib.setSessionAttr	J2EELib.setSessionAttr
SysLib.displayMsgNum	ConverseLib.displayMsgNum
SysLib.clearScreen	ConverseLib.clearScreen
SysLib.fieldInputLength	ConverseLib.fieldInputLength
SysLib.pageEject	ConverseLib.pageEject
SysLib.validationFailed	ConverseLib.validationFailed
SysLib.getVAGSysType	VGLib.getVAGSysType
SysLib.connectionService	VGLib.connectionService
SysVar.systemGregorianCalendarFormat	VGVar.systemGregorianCalendarFormat
SysVar.systemJulianDateFormat	VGVar.systemJulianDateFormat
SysVar.currentDate	VGVar.currentGregorianCalendarDate
SysVar.currentFormattedDate	VGVar.currentFormattedGregorianCalendarDate
SysVar.currentFormattedJulianDate	VGVar.currentFormattedJulianDate
SysVar.currentFormattedTime	VGVar.currentFormattedTime
SysVar.currentJulianDate	VGVar.currentJulianDate
SysVar.currentShortDate	VGVar.currentShortGregorianCalendarDate
SysVar.currentShortJulianDate	VGVar.currentShortJulianDate
SysVar.currentTime	DateTimeLib.currentTime
SysVar.currentTimeStamp	DateTimeLib.currentTimeStamp
SysVar.handleHardIOErrors	VGVar.handleHardIOErrors
SysVar.handleSysLibErrors	VGVar.handleSysLibraryErrors
SysVar.handleOverflow	VGVar.handleOverflow

*Table 1. Changed variable and function names from the SysLib and SysVar libraries (continued)*

Before migration	After migration
SysVar.mqConditionCode	VGVar.mqConditionCode
SysVar.sqlerrd	VGVar.sqlerrd
SysVar.sqlerrmc	VGVar.sqlerrmc
SysVar.sqlIsolationLevel	VGVar.sqlIsolationLevel
SysVar.sqlWarn	VGVar.sqlWarn
SysVar.commitOnConverse	ConverseVar.commitOnConverse
SysVar.eventKey	ConverseVar.eventKey
SysVar.printerAssociation	ConverseVar.printerAssociation
SysVar.segmentedMode	ConverseVar.segmentedMode
SysVar.validationMsgNum	ConverseVar.validationMsgNum

- The migration tool changes the way dates, times and timestamps are specified. Following are some examples:

*Table 2. Changes to dates, times, and timestamps*

Before migration	After migration
dateFormat = "yy/mm/dd"	dateFormat = "yy/MM/dd"
dateFormat = "YYYY/MM/DD"	dateFormat = "yyyy/MM/dd"
dateFormat = "YYYY/DDD"	dateFormat = "yyy/DDD"
timeFormat = "hh:mm:ss"	timeFormat = "HH:mm:ss"

- The migration tool sets the property `HandleHardIOErrors` to `no` for all migrated libraries, programs, and `PageHandlers` for which that property is not specified.

After you use the EGL V6.0 migration tool, make sure that none of your identifiers (such as variable or part names) begins with the at sign (@), which is now used as the first character of a complex property. Also, if you migrated code from VisualAge Generator, you may need to change `StrLib.compareStr` to `VGLib.compareBytes`, `StrLib.copyStr` to `VGLib.copyBytes`, and `StrLib.concatenate` to `VGLib.concatenateBytes`; but a change is needed only when the last argument in the existing invocation is a numeric value, as evidenced by an error message that indicates a problem with the argument's primitive type.

#### Related tasks

"Migrating EGL code to EGL V6.0 iFix" on page 103

#### Related concepts

"Setting EGL-to-EGL migration preferences" on page 112

"Changes to properties during EGL-to-EGL migration"

#### Related reference

"EGL reserved words" on page 581

## Changes to properties during EGL-to-EGL migration

The migration tool makes significant changes to the way properties are specified. Following is a summary of these changes:

- The migration tool renames properties whose names have changed in the EGL V6.0 iFix. Following is a list of the renamed properties:

*Table 3. Renamed properties*

Before migration	After migration
action	actionFunction
boolean	isBoolean
getOptions	getOptionsRecord
msgDescriptor	msgDescriptorRecord
onPageLoad	onPageLoadFunction
openOptions	openOptionsRecord
putOptions	putOptionsRecord
queueDescriptor	queueDescriptorRecord
range	validValues
rangeMsgKey	validValuesMsgKey
selectFromList	selectFromListItem
sqlVar	sqlVariableLen
validator	validatorFunction
validatorMsgKey	validatorFunctionMsgKey
validatorTable	validatorDataTable
validatorTableMsgKey	validatorDataTableMsgKey

- The migration tool adds double quotes to property values used as string literals.

**Before migration:**

```
{ alias = prog }
```

**After migration:**

```
{ alias = "prog" }
```

The following properties are affected:

- alias
- column
- currency
- displayName
- fileName
- fillCharacter
- help
- helpKey
- inputRequiredMsgKey
- minimumInputMsgKey
- msgResource
- msgTablePrefix
- pattern
- queueName
- rangeMsgKey
- tableNames
- title

- typeChkMsgKey
- validatorMsgKey
- validatorTableMsgKey
- value
- view
- The migration tool replaces parentheses with square brackets when specifying array literals as values for properties.
  - formSize
  - keyItems
  - outline
  - pageSize
  - position
  - range
  - screenSize
  - screenSizes
  - tableNames
  - tableNameVariables
  - validationBypassFunctions
  - validationBypassKeys
- For properties that take array literals, the migration tool puts single element array literals in brackets to specify that an array with only one element is still an array. The migration tool uses double sets of brackets for properties that take arrays of arrays.

**Before migration:**

```
{ keyItems = var, screenSizes = (24, 80), range = (1, 9) }
```

**After migration:**

```
{ keyItems = ["var"], screenSizes = [[24, 80]], range = [[1, 9]] }
```

- The migration tool uses the keyword **this** instead of a variable name when overriding properties for a specific element in an array.

**Before migration:**

```
Form myForm type TextForm
  fieldArray char(10)[5] { fieldArray[1] {color = red } };
end
```

**After migration:**

```
Form myForm type TextForm
  fieldArray char(10)[5] { this[1] {color = red } };
end
```

- The migration tool changes references to parts, functions, and fields, adding quotes and brackets where appropriate.

**Before migration:**

```
{ keyItems = (item1, item2) }
```

**After migration:**

```
{ keyItems = ["item1", "item2"] }
```

The following properties are affected by the migration tool in this way:

- action
- commandValueItem
- getOptions



- helpForm
  - inputForm
  - inputPageRecord
  - inputRecord
  - keyItem
  - keyItems
  - lengthItem
  - msgDescriptorRecord
  - msgField
  - numElementsItem
  - onPageLoadFunction
  - openOptionsRecord
  - putOptionsRecord
  - queueDescriptorRecord
  - redefines
  - selectFromListItem
  - tableNameVariables
  - validationBypassFunctions
  - validatorFunction
  - validatorDataTable
- The migration tool assigns a default value of yes to any boolean properties that were specified but not assigned a value.

**Before migration:**

```
{ isReadOnly }
```

**After migration:**

```
{ isReadOnly = yes }
```

The following properties are affected by the migration tool in this way:

- addSpaceForSOSI
- allowUnqualifiedItemReferences
- boolean
- bypassValidation
- containerContextDependent
- currency
- cursor
- deleteAfterUse
- detectable
- fill
- helpGroup
- includeMsgInTransaction
- includeReferencedFunctions
- initialized
- inputRequired
- isDecimalDigit
- isHexDigit
- isNullable

- isReadOnly
  - lowerCase
  - masked
  - modified
  - needsSOSI
  - newWindow
  - numericSeparator
  - openQueueExclusive
  - pfKeyEquate
  - resident
  - runValidatorFromProgram
  - segmented
  - shared
  - sqlVar
  - upperCase
  - wordWrap
  - zeroFormat
- The migration tool splits the **currency** property into two properties: **currency** and **currencySymbol**. The following table gives some examples of how the migration tool changes the **currency** property.

*Table 4. Changes to the **currency** property*

Before migration	After migration
{ currency = yes }	{ currency = yes }
{ currency = no }	{ currency = no }
{ currency = "usd" }	{ currency = yes, currencySymbol = "usd" }

- The migration tool changes the values of the **dateFormat** and **timeFormat** properties to be case sensitive. For more information, see *Date, time, and timestamp format specifiers*.
- If the **Add qualifiers to enumeration property values** check box is checked in the preferences menu, the migration tool adds the type of value to the value of the property.

Before migration:

```
color = red
outline = box
```

After migration:

```
color = ColorKind.red
outline = OutlineKind.box
```

This change affects the following properties:

- align
- color
- deviceType
- displayUse
- highlight
- indexOrientation
- intensity

- outline
- protect
- selectType
- sign
- The migration tool changes the values of the **tableNames** property to be an array of arrays of strings. Each array of strings must have either one or two elements. The first element is the table name, and the second element, if present, is the table label. The following table gives some examples of how the migration tool changes the **tableNames** property.

Table 5. Changes to the **tableNames** property

Before migration	After migration
{ tableNames = (table1, table2) }	{ tableNames = [{"table1"}, {"table2"}] }
{ tableNames = (table1 t1, table2) }	{ tableNames = [{"table1", "t1"}, {"table2"}] }
{ tableNames = (table1 t1, table2 t2) }	{ tableNames = [{"table1", "t1"}, {"table2", "t2"}] }

- The migration tool changes the values of the **tableNameVariables** property in the same way as it changes the values of the **tableNames** property.
- The migration tool changes the values of the **defaultSelectCondition** property to be of type `sqlCondition`.

**Before migration:**

```
{ defaultSelectCondition =
  #sql{
    hostVar02 = 4
  }
}
```

**After migration:**

```
{ defaultSelectCondition =
  #sqlCondition{ // no space between #sqlCondition and the brace
    hostVar02 = 4
  }
}
```

- The migration tool replaces the NULL value of the **fillCharacter** to the empty string value "".

**Related tasks**

"Migrating EGL code to EGL V6.0 iFix" on page 103

**Related concepts**

"EGL-to-EGL migration" on page 104

"Setting EGL-to-EGL migration preferences"

"Date, time, and timestamp format specifiers" on page 46

## Setting EGL-to-EGL migration preferences

You can set preferences that control how the EGL V6.0 migration tool converts EGL source code. For more information about the migration tool, see *EGL-to-EGL migration*. Set the preferences for the migration tool as follows:

1. Click **Window > Preferences**.
2. Expand **EGL**.
3. Click **EGL V6.0 Migration Preferences**.

**Note:** If you can not find **EGL V6.0 Migration Preferences**, enable the EGL V6.0 Migration capability. See *Enabling EGL Capabilities*.

4. Choose how to resolve a naming conflict with a new reserved word by clicking a radio button:
  - **Add prefix** sets the migration tool to add a prefix to any words in the source code that are now reserved words. In the text box by this radio button, type the prefix you would like the migration tool to add to the changed word.
  - **Add suffix** sets the migration tool to add a suffix to any words in the source code that are now reserved words. In the text box by this radio button, type the suffix you would like the migration tool to add to the changed word.
5. To add a qualifier to the values of properties that have a finite list of possible values, select the **Add qualifiers to enumeration property values** check box. If this box is checked, the migration tool will add the type of value to the value name.
6. To add comments to the files changed by the migration tool, choose an option under **Logging Level**.
  - **Add comments to all files processed by the migration tool** sets the migration tool to add a comment to each file it processes, even if it makes no changes to that file.
  - **Add comments to all files that are changed by the migration tool** sets the migration tool to add a comment to only the files it changes.
  - **Do not add comments to files** sets the migration tool to not add any comments to the files it processes.
  - **Add to beginning of file** sets the migration tool to add comments to the beginning of files.
  - **Add to end of file** sets the migration tool to add comments to the end of files.
7. To write a list of the processed files to the file named V60MigrationLog.txt, select the **Append migration results to log file per project** check box.
8. Click **Apply**.
9. Click **OK**.

#### **Related tasks**

"Enabling EGL capabilities" on page 124

"Migrating EGL code to EGL V6.0 iFix" on page 103

#### **Related concepts**

"EGL-to-EGL migration" on page 104



---

## Setting up the environment

---

### Setting EGL preferences

Set the basic EGL preferences as follows:

1. Click **Window > Preferences**.
2. When a list is displayed, click **EGL** to display the EGL screen.
3. Select or clear the check box for **VisualAge Generator Compatibility**. Your choice affects what options are available at development time, as described in *Compatibility with VisualAge Generator*.
4. In the **Encoding** list box, select the character-encoding set that will be used when you create new EGL build (.eglbld) files. The setting has no effect on existing build files. The default value is UTF-8.
5. In the **User ID** text box, specify the user ID for accessing the remote build machine, if any. The build descriptor option **destUserID** takes precedence, and both that option and the preference value take precedence over the master build descriptor option **destUserID**.
6. In the **Password** text box, specify the password for accessing the remote build machine, if any. The build descriptor option **destPassword** takes precedence, and both that option and the preference value take precedence over the master build descriptor option **destPassword**.
7. Select or clear the check boxes under **Default EGL Web Project Feature Choices**. These check boxes determine which features are enabled by default in new EGL Web projects.
8. Click **Apply**.

To set other EGL preferences, see the list of related tasks at the bottom of this page. When you are finished setting preferences, click **OK**.

#### Related concepts

"Build" on page 411

"Compatibility with VisualAge Generator" on page 532

#### Related tasks

"Setting the default build descriptors" on page 118

"Setting preferences for the EGL debugger" on page 116

"Setting preferences for source styles" on page 119

"Setting preferences for SQL database connections" on page 121

"Setting preferences for SQL retrieve" on page 123

### Setting preferences for text

To change how text is displayed in the EGL editor, do as follows:

1. Click **Window > Preferences**.
2. When a list of preferences is displayed, expand **Workbench**, then click **Colors and Fonts**. The Colors and Fonts pane is displayed.
3. Expand **EGL** and **Editor**, then click **EGL Editor Text Font**.
4. To choose from a list of fonts and colors, click the **Change** button, then do as follows:

- a. To change the your font preferences, select a font, font style, and size from the scrollable lists.
  - b. To change your color preference, select a color from the drop-down list.
  - c. Select the **Strikeout** check box if you want a line to run through the middle of the text.
  - d. Select the **Underline** check box if you want a line under the text.
  - e. You can see a preview of your selections in the Sample box. When you are finished making your selections, click **OK**.
5. To use the default operating system font, click the **Use System Font** button.
  6. To use the default workbench font, click the **Reset** button.
  7. To set the font for all editors (not just the EGL editor) to the default workbench font, click the **Restore Defaults** button.
  8. To save your changes, click **Apply** or (if you are finished setting preferences) click **OK**.

#### Related tasks

"Setting EGL preferences" on page 115

## Setting preferences for the EGL debugger

To set preferences for the EGL debugger, follow these steps:

1. Click **Window > Preferences**.
2. When a list is displayed, expand **EGL** and click **Debug**.
3. Clear or select the check box labeled **Prompt for SQL user ID and password when needed**.  
For details on your choice, see *EGL debugger*.
4. Clear or select the check box labeled **Set systemType to DEBUG**.  
For details on your choice, see *EGL debugger*.
5. Set the initial values for sysVar.terminalID, sysVar.sessionID, and sysVar.userID. If you do not specify values, each defaults to your user ID on Windows 2000/NT/XP or Linux®.

6. In the **Remote User** and **Remote Password** fields, enter the user ID and password to be used for remote calls while debugging.

The user ID and password used for remote calls while debugging are separate from the user ID and password used to access an SQL database. To set the user ID and password for SQL database connections, see *Setting preferences for SQL database connections*.

7. Set the EGL Debugger Port value. The default is 8345.
8. Select the type of character encoding to use when processing data during a debugging session. The default is the local system's file encoding. For details on your choice, see *Character encoding options for the EGL debugger*.
9. To specify external Java classes for use when the debugger runs, modify the class path. You might need extra classes, for example, to support MQSeries, JDBC drivers, or Java access functions.

The class path additions are not visible to the WebSphere Application Server test environment; but you can add to that environment's classpath by working on the Environment tab of the server configuration.

Use the buttons to the right of the Class Path Order box:

- To add a project, JAR file, directory, or variable, click the appropriate button: **Add Project**, **Add JARs**, **Add Directory**, or **Add Variable**.

- To remove an entry, select it and click **Remove**.
  - To move an entry in a list of two or more entries, select the entry and click **Move Up** or **Move Down**.
10. To restore the default settings, click **Restore Defaults**.
  11. To save your changes, click **Apply** or (if you are finished setting preferences) click **OK**.

#### Related concepts

“Compatibility with VisualAge Generator” on page 532

“EGL debugger” on page 369

“Character encoding options for the EGL debugger”

#### Related tasks

“Setting preferences for SQL database connections” on page 121

#### Related reference

“sessionID” on page 1071

“terminalID” on page 1075

“userID” on page 1076

### Character encoding options for the EGL debugger

The EGL debugger allows you to specify the type of character encoding to use while debugging. Character encoding controls how the debugger represents character and numeric data internally, how it compares character data, and how it passes parameters to remote programs, files, and databases. To change these options, see *Setting preferences for the EGL debugger*.

The EGL debugger supports two different types of character encoding: the default encoding on your local system and EBCDIC. The default character encoding for the EGL debugger is the same as your local system’s default encoding.

- When the default character encoding is selected, the debugger represents CHAR, DBCHAR, MBCHAR, DATE, TIME, INTERVAL, NUM, and NUMC variables in the default format, typically ASCII. Comparisons between character variables use the ASCII collating sequence. Data must be converted to host format when calling remote programs and when accessing remote files and databases.

If you choose this setting and do not specify a conversion table, the debugger chooses an appropriate conversion table when you call a remote program or access a remote file or database. For more information on conversion tables, see *Data conversion*.

- When EBCDIC character encoding is used, the debugger represents CHAR, DBCHAR, MBCHAR, DATE, TIME, and INTERVAL variables with EBCDIC encoding. NUM and NUMC variables are represented in host numeric format. Comparisons between character variables use the EBCDIC collating sequence. Data does not need to be converted to host format when calling remote programs or when accessing remote files and databases, but data is converted to the appropriate Java or ASCII format when making SQL calls or calls to local C++ routines. EBCDIC encoding is available in several languages.

If you choose EBCDIC character encoding and do not specify a conversion table, the debugger does not use a conversion table when you call a remote program or access a remote file or database. The program name, library name, and any passed parameters are encoded according to EBCDIC character encoding.



If the selected character encoding is unsupported by your Java Runtime Environment, you will see a warning message when the debugger starts. If you choose to continue debugging, the debugger will return to the default encoding type.

You can not change character encoding during a debugging session. You must restart the debugger for a change in character encoding to take effect.

#### **Related concepts**

“EGL debugger” on page 369

#### **Related tasks**

“Setting preferences for the EGL debugger” on page 116

#### **Related reference**

“Data conversion” on page 558

## **Setting the default build descriptors**

For an overview of the default build descriptor and the build descriptor precedence rules, see *Generation in the workbench*.

To specify a preference for build descriptors at the Workbench level, do as follows:

1. Click **Window > Preferences**.
2. When a list is displayed, expand **EGL** and click **Default Build Descriptor**.
3. Select the Debug build descriptor and the Target system build descriptor.
4. Click **Apply**, then click **OK**.

To specify a preference for build descriptors at the file, folder, package, or project level, do as follows:

1. Right-click on the level of interest (for example, on the file or folder name) and, from the context menu, click **Properties**.
2. Select **EGL Default Build Descriptors**.
3. Select the Debug build descriptor and the Target system build descriptor.
4. Click **OK**.

#### **Related concepts**

“Generation in the workbench” on page 416

## **Setting preferences for the EGL editor**

To specify the EGL editor preferences, do as follows:

1. Click **Window > Preferences**.
2. When a list is displayed, expand **EGL** and click **Editor**.
3. To display line numbers when you review an EGL file, select the **Show line numbers** check box. To clear the line numbers, clear the check box. The file itself is not affected.
4. To show red underlines wherever errors are found in the source code, select the **Annotate errors in text** check box. To clear those underlines, clear the check box. The file itself is not affected.
5. To show a red error indicator in the right margin of the editor (overview ruler) whenever an error is found in the source code, select the **Annotate errors in**

**overview ruler** check box. Clicking on the error indicator will take you to the location of the error in the source code. To clear the error indicator, clear the check box. The file itself is not affected.

6. To specify source styles, follow the process described in *Setting preferences for source styles*.
7. To add, remove, and customize templates for use in content assist, follow the process described in *Setting preferences for templates*.
8. To change how text is displayed, follow the process described in *Setting preferences for text*.

#### Related tasks

"Setting preferences for source styles"

"Setting preferences for templates"

"Setting preferences for text" on page 115

#### Related reference

"Content assist in EGL" on page 577

## Setting preferences for source styles

You can change how EGL code is displayed in the EGL editor:

1. Click **Window > Preferences**
2. When a list of preferences is displayed, expand **EGL** and **Editor**, then click **Source Styles**.
3. To select the color that you want to appear behind the source type, click the **Custom** radio button in the Background color box. Click the button next to the Custom label. A color palette displays. Select a color, then click **OK**.
4. In the Foreground box, select a type of text, then click the **Color** button. A color palette displays. Select a color, then click **OK**.
5. Select the **Bold** check box if you want to make the type bold.
6. To save your changes, click **Apply** or (if you are finished setting preferences) click **OK**.

#### Related tasks

"Setting EGL preferences" on page 115

## Setting preferences for templates

Do as follows to add, remove, or customize the templates that are displayed when you request content assist in the EGL editor:

1. Click **Window > Preferences**.
2. When a list of preferences is displayed, expand **EGL** and **Editor**, then click **Templates**. A list of templates is displayed.

**Note:** As in other applications on Windows 2000/NT/XP, you can click an entry to select it; can use **Ctrl-click** to select or unselect an entry without affecting other selections; and can use **Shift-click** to select a set of entries that are contiguous to the entry you last clicked.

3. To make a template available in the EGL editor, select the check box to the left of a template name. To make all the listed templates available, click **Enable All**. Similarly, to make a template unavailable, clear the related check box; and to make all the listed templates unavailable, click **Disable All**.
4. To create a new template, do as follows--

- a. Click **New**
- b. When the New Template dialog is displayed, specify both a name and a description because a template is guaranteed to be displayed in a content-assist list only if the combination of name and description are unique across all templates.

**Note:** If the first word used in the template is an EGL keyword (such as Function), the template is available when you request content assist in the EGL editor, but only when the on-screen cursor is at a place where the word is valid. Similarly, if you type a prefix, then request content assist, all templates beginning with that prefix are available provided the on-screen cursor is in a position where that template is syntactically allowed. For example, type "fun" to request function templates. If you do not type either a prefix or the full first word, you will not see any templates when you request content assist.

- c. In the **Pattern** field, type the template itself:
  - Type any text that you wish to display
  - To place a preexisting variable at the on-screen cursor position, click **Insert Variable**, then double-click a variable. When you insert the template in the EGL editor, each of those variables resolves to the appropriate value.
  - To create a custom variable, type a dollar sign (\$) followed by a left brace ({}), a string, and a right brace ({}), as in this example:  
`${variable}`  
 You may find it easier to insert a preexisting variable and change the name for your own use.  
 When you insert a custom template in the EGL editor, each variable is underlined to indicate that a value is required.
  - To complete the task, click **OK** and, at the templates screen, click **Apply**.
5. To review an existing template, click on the listed entry and review the Preview box.
6. To edit an existing template, click on the listed entry, then click **Edit**. Interact with the Edit Template dialog as you did with the New Template dialog.
7. To remove an existing template, click on the listed entry, then click **Remove**. To remove multiple templates, use the Windows 2000/NT/XP convention for selecting multiple list entries, then click **Remove**.
8. To import a template from an XML file, click **Import** at the right of the template list and follow the browse mechanism to specify the location of the file.
9. To export a template to an XML file, click **Export** at the right of the template list and follow the browse mechanism to specify the location of the new file. To export multiple templates, use the Windows 2000/NT/XP mechanism for selecting multiple list entries, then click **Export**.
10. To export all the listed templates to an XML file, click **Export All** and follow the browse mechanism to specify the location of the file.
11. To save your changes, click **Apply**. To return to the template list that was in effect at installation time, click **Restore Defaults**.

#### Related tasks

"Using the EGL templates with content assist" on page 131

## Setting preferences for SQL database connections

You use the page for SQL database connections for these reasons:

- You can enable declaration-time and debug-time access to a database that is accessed outside of J2EE.
- Also, you can set a value for the build descriptor option `sqlJNDIName`, which specifies a name to which the default datasource is bound in the JNDI registry; for example, `java:comp/env/jdbc/MyDB`. That option is included in the build descriptor that is created for you in the following situation:
  - You use the EGL Web Project Wizard, as described in *Creating a project to work with EGL*; and
  - When working in that wizard, you request that a build descriptor be created.

Do as follows:

1. Click **Window > Preferences**
2. When a list of preferences is displayed, expand **EGL**, then click **SQL Database Connections**.
3. In the **Connection URL** field, type the URL used to connect to the database through JDBC:

- For IBM DB2 Universal Driver and IBM DB2 APP DRIVER for Windows, the URL is `jdbc:db2:dbName` (where *dbName* is the database name)
- For the Oracle JDBC thin client-side driver, the URL varies by database location. If the database is local to your machine, the URL is `jdbc:oracle:thin:dbName` (where *dbName* is the database name). If the database is on a remote server, the URL is `jdbc:oracle:thin:@host:port:dbName` (where *host* is the host name of the database server, *port* is the port number, and *dbName* is the database name)
- For the Informix JDBC NET driver, the URL is as follows (with the lines combined into one)--

```
jdbc:informix-sqli://host:port  
/dbName:informixserver=servername;  
user=userName;password=password
```

*host*

Name of the machine on which the database server resides

*port*

Port number

*dbName*

Database name

*serverName*

Name of the database server

*userName*

Informix user ID

*password*

Password associated with the user ID

- For the DataDirect SequeLink JDBC Driver for SQL Server, the URL is as follows (with the lines combined into one)--

```
jdbc:sequelink://host:port;  
SelectMethod=cursor;DatabaseName=dbName
```

*host*

Name of the machine on which the database server resides

*port*

Port number

*dbName*

Database name

- For the Microsoft® JDBC Driver for SQL Server, the URL is as follows (with the lines combined into one)--

```
jdbc:microsoft:sqlserver://host:port;  
SelectMethod=cursor;DatabaseName=dbName
```

*host*

Name of the machine on which the database server resides

*port*

Port number

*dbName*

Database name

4. In the **Database** field, type the name of the database.
5. In the **User ID** field, type the user ID for the connection.
6. In the **Password** field, type the password for the user ID.
7. In the **Database vendor type** field, select the database product and version that you are using for your JDBC connection.
8. In the **JDBC driver** field, select the JDBC driver that you are using for your JDBC connection.
9. In the **JDBC driver class** field, type the driver class for the driver you selected:
  - For IBM DB2 Universal Driver, the driver class is  
com.ibm.db2.jcc.DB2Driver
  - For IBM DB2 APP DRIVER for Windows, the driver class is  
COM.ibm.db2.jdbc.app.DB2Driver
  - For the Oracle JDBC thin client-side driver, the driver class is  
oracle.jdbc.driver.OracleDriver
  - For the Informix JDBC NET driver, the driver class is  
com.informix.jdbc.IfxDriver
  - For the DataDirect SequeLink JDBC Driver for SQL Server, the driver class is  
com.ddtek.jdbc.sqlserver.SQLServerDriver
  - For the Microsoft JDBC Driver for SQL Server, the driver class is  
com.microsoft.jdbc.sqlserver.SQLServerDriver
  - For other driver classes, refer to the documentation for the driver
10. In the **class location** field, type the fully qualified filename of the \*.jar or \*.zip file that contains the driver class:
  - For IBM DB2 Universal Driver, type the fully qualified filenames to the db2jcc.jar and db2jcc\_license\_cu.jar files
  - For IBM DB2 APP DRIVER for Windows, type the fully qualified filename to the db2java.zip file; for example, d:\sql11b\java\db2java.zip
  - For the Oracle THIN JDBC DRIVER, type the fully qualified pathname to the ojdbc14.jar file; for example, d:\0ra81\jdbc\lib\ojdbc14.jar or, if you require Oracle trace, ojdbc14\_g.jar
  - For the Informix JDBC NET driver, type the fully qualified filename to the ifxjdbc.jar file
  - For the DataDirect SequeLink JDBC Driver for SQL Server, type the fully qualified filenames to the base.jar, util.jar, and sqlserver.jar files

- For the Microsoft JDBC Driver for SQL Server, type the fully qualified filenames to the msbase.jar, msutil.jar, and mssqlserver.jar files
  - For other driver classes, refer to the documentation for the driver
11. In the **Connection JNDI name** field, specify the database used in J2EE. The value is the name to which the datasource is bound in the JNDI registry; for example, java:comp/env/jdbc/MyDB. As noted earlier, this value is assigned to the option **sqlJNDIName** in the build descriptor that is constructed automatically for a given EGL Web project.
  12. If you are accessing DB2 UDB and specify a value in the **Secondary authentication ID** field, the value is used in the SET CURRENT SQLID statement used by EGL at validation time. The value is case-sensitive.

You can clear or apply preference settings:

- To restore default values, click **Restore Defaults**.
- To apply preference settings without exiting the preferences dialog, click **Apply**.
- If you are finished setting preferences, click **OK**.

#### Related tasks

"Creating an EGL Web project" on page 127

"Setting EGL preferences" on page 115

#### Related reference

"dbms" on page 471

"sqlJNDIName" on page 492

## Setting preferences for SQL retrieve

At EGL declaration time, you can use the SQL retrieve feature to create an SQL record from the columns of an SQL table. For an overview, see *SQL support*.

To set preferences for the SQL retrieve feature, do as follows:

1. Click **Window > Preferences**, then expand **EGL** and click **SQL Retrieve**
2. Specify rules for creating each structure item that is created by the SQL retrieve feature:
  - a. To specify the EGL type to use when creating a structure item from an SQL character data type, click one of the following radio buttons:
    - **Use EGL type string** (the default) maps SQL char data types to EGL string data types
    - **Use EGL type limited-length string** maps SQL char data types to EGL limited-length string data types
    - **Use EGL type char** maps SQL char data types to EGL char data types
    - **Use EGL type mbChar** maps SQL char data types to EGL mbChar data types
    - **Use EGL type Unicode** maps SQL char data types to EGL Unicode data types
  - b. To specify the case of the structure item name, click one of the following radio buttons:
    - **Do not change case** (the default) means that the case of the structure item name is the same as the case of the related table column name
    - **Change to lower case** means that the structure item name is a lower-case version of the table column name

- **Change to lower case and capitalize first letter after underscore** also means that the structure item name is a lower-case version of the table column name, except that a letter in the structure item name is rendered in uppercase if, in the table column name, the letter immediately follows an underscore
- c. To specify how the underscores in the table column name are reflected in the structure item name, click one of the following radio buttons:
  - **Do not change underscores** (the default) means that underscores in the table column name are included in the structure item name
  - **Remove underscores** means that underscores in the table column name are not included in the structure item name
- 3. If you want new SQL records to have the key item property set, select the **Retrieve primary key information from the system catalog** check box.

#### Related concepts

"SQL support" on page 277

#### Related tasks

"Retrieving SQL table data" on page 299

"Setting EGL preferences" on page 115

"Setting preferences for SQL database connections" on page 121

#### Related reference

"Informix and EGL" on page 299

---

## Enabling EGL capabilities

In order to access EGL functionality, the EGL capabilities must be enabled. The following EGL capabilities are available:

### EGL Development

Consists of all functionality related to developing and debugging EGL applications.

### EGL V6.0 Migration

Consists of all functionality related to converting EGL 5.1.2 and 6.0 source code to comply with the EGL V6.0 iFix.

### VisualAge Generator to EGL Migration

Consists of all functionality related to migrating existing VisualAge Generator code to EGL code.

To enable EGL capabilities, do as follows:

1. Click **Window > Preferences**.
2. When a list of preferences is displayed, expand **Workbench**, then click **Capabilities**. The Capabilities pane is displayed.
3. If you would like to receive a prompt when a feature is first used that requires an enabled capability, select the check box for **Prompt when enabling capabilities**.
4. Expand the **EGL Developer** capability folder.
5. Select the check box for the desired EGL capabilities. Alternately, you can select the **EGL Developer** capability folder to enable all of the capabilities that folder contains.
6. To set the list of enabled capabilities back to its state at product install time, click the **Restore Defaults** button.

7. To save your changes, click **Apply**, then click **OK**.

**Note:** Enabling EGL capabilities will automatically enable any other capabilities that are required to develop and debug EGL applications.

**Related tasks**

“Setting EGL preferences” on page 115





---

## Beginning code development

---

### Creating a project

#### Creating an EGL project

For an overview of how to organize your work, see *EGL projects, packages, and files*.

To set up a new EGL project, do as follows:

1. In the Workbench, do either of the following steps:

- Click **File > New > Project**; or
- Right-click, then click **New > Project**.

The New Project wizard opens.

2. Expand **EGL**, then click **EGL Project**. Click **Next**. The **New EGL project** wizard is displayed.

**Note:** If EGL Project is not available, check the **Show All Wizards** check box.

3. In the **Project name** field, type a name for the project. By default, the project is placed in your workspace, but you can click **Browse** and choose a different location.
4. Select how to specify a build descriptor, which is the part that directs processing at generation time:
  - **Create new project build descriptor(s) automatically** means that EGL provides build descriptors and writes them to a build file (extension .eglbld) that has the same name as the project.

To specify some of the values in those build descriptors, click **Options**. To change those values later, change the build file that is created for you.  
For further details, see *Specifying database options at project creation*.
  - **Use build descriptor specified in EGL preference** means that EGL points to a build descriptor that you created and identified as an EGL preference.
  - **Select existing build descriptor** allows you to specify a build descriptor from those that are available in your workspace.
5. In most cases, click **Finish**. If you click **Next**, however, you can specify other source folders and projects to reference from the project you are creating. When you have finished selecting other source folders and projects, click **Finish**.

#### Related concepts

"Build descriptor part" on page 383

"EGL projects, packages, and files" on page 15

#### Related tasks

"Specifying database options at project creation" on page 129

"Setting preferences for SQL database connections" on page 121

#### Creating an EGL Web project

For an overview of how to organize your work, see *EGL projects, packages, and files*.

To set up a new EGL Web project, do as follows:

1. In the Workbench, do either of the following steps:

- Click **File > New > Project**; or
- Right-click, then click **New > Project**.

The New Project wizard opens.

- Expand **EGL**, then click **EGL Web Project**. Click **Next**. The **New EGL Web project** wizard is displayed.
- In the **Project name** field, type a name for the project. By default, the project is placed in your workspace; but you can click **Browse** and choose a different location.
- Select how to specify a build descriptor, which is the part that directs processing at generation time:
  - **Create new project build descriptor(s) automatically** means that EGL provides build descriptors and writes them to a build file (extension .eglbld) that has the same name as the project.  
To specify some of the values in those build descriptors, click **Options**. To change those values later, change the build file that is created for you.  
For further details, see *Specifying database options at project creation*.
  - **Use build descriptor specified in EGL preference** means that EGL points to a build descriptor that you created and identified as an EGL preference.
  - **Select existing build descriptor** allows you to specify a build descriptor from those that are available in your workspace.
- If you requested that a build descriptor be created automatically, you can place a value in the **JNDI Name for SQL Connection** field. The effect is to assign the name to which the default data source is bound in the JNDI registry at debug or generation time. (An example value is `java:comp/env/jdbc/MyDB`.) Your selection assigns a value to the build descriptor option `sqlJNDIName`. If the **JNDI Name for SQL Connection** field is already populated, the value was obtained from a Workbench preference, as described in *Setting preferences for SQL database connections*.
- In most cases, click **Finish**. To do additional customization (as is possible for any Web project), configure the J2EE settings at the bottom of the dialog. Optionally, you can click **Hide Advanced** to conceal the J2EE settings.
- To configure the Web project features, click **Next**. The following EGL-related features are available on the Features page:
  - **EGL support with JSF** is selected by default. This feature lets the EGL Web project use JavaServer Faces components to display information on Web pages and receive input from Web pages.
  - **EGL support with JSF Component Interfaces** allows you to call Java functions recognized by the JavaServer Faces components, giving you more direct control over their appearance and behavior. See *Accessing a JSF component from a pageHandler*.
  - **EGL support with Legacy Web Transaction** adds support for Web Transactions to the project.

The other features in this list do not affect EGL directly.

You can add any of these features later by right-clicking the project in the Project Explorer view and clicking **Properties**. Then click **Web Project Features** and use the check boxes to add features. You can not remove features from a project.

- To select a default page template for the Web project, click **Next**. Select a page template, then click **Finish**.

**Related concepts**

“Build descriptor part” on page 383

“EGL projects, packages, and files” on page 15

**Related tasks**

“Specifying database options at project creation”

“Setting preferences for SQL database connections” on page 121

“Accessing a JSF component from a pageHandler” on page 232

“Configuring a project to run Web transactions” on page 164

**Related reference**

“sqlJNDIName” on page 492

## Specifying database options at project creation

To assign option values in the build descriptor that is created automatically by EGL, work at the **Project Build Options** dialog. For details on how to display the dialog, see *Creating a project to work with EGL*.

To accept the database-connection information that was specified in preferences, click the check box.

The next table shows each on-screen label and the related build descriptor option.

Label	Build descriptor option
Database type	dbms
Database JDBC driver	sqlJDBCDriverClass
Database name	sqlJNDIName (for J2EE output) or sqlDB (for non-J2EE output)

**Related concepts**

“Build descriptor part” on page 383

**Related tasks**

“Creating an EGL Web project” on page 127

**Related reference**

“Build descriptor options” on page 464

---

## Creating an EGL source folder

Once you create a project in the workbench, you can create one or more folders within that project to contain your EGL files.

To create a folder for grouping EGL files, do as follows:

1. In the workbench, click **File > New > EGL Source Folder**.
2. Select the project that will contain the EGL folder. In the Folder name field, type the name of the EGL folder, for example myFolder.
3. Click the **Finish** button.

**Related concepts**

“EGL projects, packages, and files” on page 15

“Introduction to EGL” on page 1

#### Related tasks

“Creating an EGL Web project” on page 127

#### Related reference

“Creating an EGL source file”

“Naming conventions” on page 778

---

## Creating an EGL package

An EGL package is a named collection of related source parts. To create an EGL package, do as follows:

1. Identify a project or folder to contain the package. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > EGL Package**.
3. Select the project or folder that will contain the EGL package. The Source Folder field may be pre-populated depending on the current selection in the Project Explorer.
4. In the Package Name field, type the name of the EGL package. See *EGL projects, packages, and files* for details on package naming conventions.
5. Click the **Finish** button.

#### Related concepts

“EGL projects, packages, and files” on page 15

“Introduction to EGL” on page 1

#### Related tasks

“Creating an EGL source folder” on page 129

“Creating an EGL Web project” on page 127

#### Related reference

“Creating an EGL source file”

---

## Creating an EGL source file

To create an EGL source file, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > EGL Source File**.
3. Select the project or folder that will contain the EGL source file. Select the package that will contain the EGL source file. In the EGL Source File Name field, type the name of the EGL source file, for example myEGLFile.
4. Click **Finish** to create the file. An extension (.egl) is automatically appended to the end of the file name. The EGL source file appears in the Project Explorer view and automatically opens in the default EGL editor.

#### Related concepts

“EGL projects, packages, and files” on page 15

“Introduction to EGL” on page 1

#### Related tasks

“Creating an EGL source folder” on page 129

“Creating an EGL Web project” on page 127

---

## Using the EGL templates with content assist

To practice using content assist, do as follows:

1. Open a new EGL file.
2. On an available line, type **P** (for PageHandler or program) and press **Ctrl + Space**.
3. When a pop-up is displayed, click an icon for the part to customize. Do either of these steps:
  - Press **Enter** to select the first icon in the list; or
  - Use the arrow keys to select another icon (for a program) and press **Enter**.The editor places a part template in your file.

4. Customize the part.

When the template is displayed, the editor highlights the first area where you need to type information; in this case, specify the part name. After you type, press **Tab** to highlight the next area where you need to type.

You can use the **Tab** key repeatedly, and this use of the key is available until you reach the end of the file or until you change your in-file position in any other way.

5. To insert a function into your program or PageHandler, type **F** (for function), then press **Ctrl + Space**. Although you can select a part template again, do as follows

- Use the arrow keys or your mouse to scroll to the end of the list
- Press **Enter** or click the word *Function*; note that the absence of an icon means that you are selecting a string rather than a part template

The ability to select a string is more useful in other contexts, such as when you want to type a variable name quickly.

6. With the cursor at the end of the word *Function*, press **Ctrl + Space** and click an icon from the list.

The editor places the function template in your file.

7. Customize the part.
8. As you develop your code, periodically press **Ctrl + Space** to understand the range of services that are provided.

### Related tasks

“Inserting code snippets into EGL and JSP files” on page 179

“Setting preferences for templates” on page 119

### Related reference

“Function part in EGL source format” on page 621

“PageHandler part in EGL source format” on page 785

“Program part in EGL source format” on page 841

---

## Keyboard shortcuts for EGL

The next table shows the keyboard shortcuts that are available in the EGL editor.

Key combination	Function
Ctrl+/ 	Comment
Ctrl+\ 	Uncomment
Ctrl+A	Select all

Key combination	Function
Ctrl+C	Copy
Ctrl+F	Find
Ctrl+H	Search
Ctrl+K	Find next
Ctrl+S	Save
Ctrl+V	Paste
Ctrl+X	Cut
Ctrl+G	Generate
Ctrl+L	Go to a specific line
Ctrl+Y	Redo
Ctrl+Z	Undo
Ctrl+Shift+A	Add an explicit SQL statement to an EGL I/O statement that has an implicit one
Ctrl+Shift+K	Find previous
Ctrl+Shift+N	Access the Open Part dialog
Ctrl+Shift+P	Construct an EGL <b>prepare</b> statement and the related <b>get</b> , <b>execute</b> , or <b>open</b> statement
Ctrl+Shift+R	Use the retrieve feature to create or overwrite items in an SQL record part
Ctrl+Shift+S	Show the current file in Project Explorer
Ctrl+Shift+V	View and validate the SQL statement that is associated with an EGL I/O statement and perform related actions
Ctrl+Shift+Z	Use the Source Assistant
Ctrl+Space	Get content assist
F3	Open the file that contains the part whose name is highlighted
Tab	Indents text to the next tab stop

---

## Developing basic EGL source code

---

### Creating an EGL dataItem part

An EGL dataItem part defines an area of memory that cannot be divided. EGL dataItem parts are contained in EGL files. To create an EGL dataItem part, do as follows:

1. Identify an EGL file to contain the dataItem part and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the dataItem part according to EGL syntax (for details, see *DataItem part in EGL source format*). You can use content assist to place an outline of the dataItem part syntax in the file.
3. Save the EGL file.

#### Related concepts

"DataItem part"

"EGL projects, packages, and files" on page 15

#### Related tasks

"Creating an EGL source file" on page 130

"Using the EGL templates with content assist" on page 131

#### Related reference

"Content assist in EGL" on page 577

"DataItem part in EGL source format" on page 566

"Naming conventions" on page 778

### DataItem part

A *dataItem part* defines an area of memory that cannot be subdivided. A dataItem part is a standalone part, unlike a structure field in a fixed structure.

A *primitive variable* is a memory area that is based on a dataItem part or on a primitive declaration such as INT or CHAR(2). You may use a primitive variable in these ways:

- As a parameter that receives data into a function or program
- As a variable in an EGL function; for instance, in an assignment statement or as an argument that passes data to another function or program

Each primitive variable has a series of properties, whether by default or as specified in either the variable or the dataItem part. For details, see *Overview of EGL properties and overrides*.

#### Related concepts

"Fixed record parts" on page 136

"Fixed structure" on page 27

"Overview of EGL properties" on page 64

"Parts" on page 19

"Record parts" on page 135

"Typedef" on page 28



### Related tasks

“Setting preferences for templates” on page 119

### Related reference

“DataItem part in EGL source format” on page 566

“EGL source format” on page 586

“Data initialization” on page 564

“Primitive types” on page 34

## Editing a dataItem part with the source assistant

To use the source assistant to edit a dataItem part, follow these steps:

1. Open an EGL source file.
2. Declare a dataItem part or select an existing dataItem part that you want to edit. The cursor must be on the dataItem part that you want to edit.  
For more information about creating a new dataItem part, see *DataItem part in EGL source format*.
3. Open the source assistant by using one of these methods:
  - Press **Ctrl + Shift + Z**.
  - Right-click on the dataItem part and then click **Source Assistant**.
4. At the top of the EGL Source Assistant window, enter the name and type of the dataItem in the **Name** and **Type** fields.
5. If the type of dataItem requires a length or a number of decimal places, enter these values in the **Length** and **Decimals** fields.  
If the type of dataItem does not require a length or a number of decimal places, these fields are unavailable.
6. Enter values for the properties of the dataItem. You can switch between different pages of properties by clicking the tabs below the **Type** field.  
For information about properties, see *Overview of EGL properties* or search for help on a particular EGL property.
7. When you are finished editing the values, click **Validate**.  
The source assistant validates the values you entered and lists any errors at the bottom of the window.
8. Correct any errors that are listed in the window and click **Validate** again.
9. When there are no more validation errors listed, click **OK**.  
The source assistant closes and updates the dataItem’s properties to match the properties you specified.
10. Save the EGL source file.

### Related concepts

“DataItem part” on page 133

“Source assist in EGL” on page 577

“Overview of EGL properties” on page 64

### Related reference

“Content assist in EGL” on page 577

“DataItem part in EGL source format” on page 566

---

## Creating an EGL record part

A record part defines a structure (a hierarchical layout of fixed-size data elements in storage) and an optional binding, which is a relationship of the record to an external data source (file, database, or message queue). EGL record parts are contained in EGL files. To create an EGL record part, do as follows:

1. Identify an EGL file to contain the record part and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the record part according to EGL syntax (for details, see *Basic record part in EGL source format*, *Indexed record part in EGL source format*, *MQ record part in EGL source format*, *Relative record part in EGL source format*, *Serial record part in EGL source format*, and *SQL record part in EGL source format*). You can use content assist to place an outline of the record part syntax in the file.
3. Save the EGL file.

### Related concepts

“EGL projects, packages, and files” on page 15  
“Record parts”

### Related tasks

“Creating an EGL source file” on page 130  
“Using the EGL templates with content assist” on page 131

### Related reference

“Basic record part in EGL source format” on page 461  
“Content assist in EGL” on page 577  
“Indexed record part in EGL source format” on page 632  
“MQ record part in EGL source format” on page 769  
“Naming conventions” on page 778  
“Relative record part in EGL source format” on page 865  
“Serial record part in EGL source format” on page 868  
“SQL record part in EGL source format” on page 877

## Record parts

A *record part* defines a sequence of fields and is the basis of a *record*, which is a variable whose allocated memory conforms to the format of the related record part.

EGL provides two categories of record parts:

### Fixed

A fixed record part defines a set of fields whose length is known at generation time. Fixed records are used primarily for accessing VSAM files, MQSeries messages queues, and other sequential files. For details, see *Fixed record parts*.

### Non-fixed

A non-fixed record part defines a set of fields whose length is not necessarily known at generation time. For details, see *Non-fixed record parts*.

### Related concepts

“DataItem part” on page 133  
“Fixed record parts” on page 136  
“Fixed structure” on page 27  
“Non-fixed record parts” on page 137  
“Parts” on page 19

“Record types and properties” on page 138  
“Resource associations and file types” on page 393  
“Typedef” on page 28

#### **Related tasks**

“Setting the default build descriptors” on page 118  
“Setting preferences for the EGL editor” on page 118

#### **Related reference**

“EGL source format” on page 586  
“Data initialization” on page 564  
“Primitive types” on page 34

## **Fixed record parts**

A fixed record part defines a sequence of data whose length is known at generation time. This kind of part is necessarily composed of a series of primitive, fixed-length fields, and each field can be substructured. A field that specifies a telephone number, for example, can be defined as follows:

```
10 phoneNumber    CHAR(10);  
20 areaCode      CHAR(3);  
20 localNumber   CHAR(7);
```

Although you can use fixed records (which are variables) for any kind of processing, their best use is for I/O operations on VSAM files, MQSeries messages queues, and other sequential files. Although you can use fixed records for accessing relational databases or for general processing (as was the case with earlier products such as VisualAge Generator), you should avoid using fixed records for those purposes in new development.

A record part of any of the following types is a fixed record part:

- DLIsegment
- indexedRecord
- mqRecord
- PSBDataRecord
- relationalRecord
- serialRecord
- VGUIRecord

In addition, a record part of any of the following types is a fixed record part if each field is preceded by a level number:

- basicRecord
- SQLRecord

You can use a fixed record in the following contexts:

- In a statement that copies data to or from a data source
- In an assignment or move statement
- As an argument that passes data to another program or function
- As a parameter that receives data into a program or function

Any relationship of a fixed record part to an external data source is determined by the type of the fixed record part and by a set of type-specific properties such as fileName. A record based on a part of type indexedRecord, for example, is used for

accessing a VSAM Key Sequenced Data Set. The relationship of a record part to a data source determines the operations that are generated when the fixed record is used in an EGL I/O statement such as **add**.

A fixed-record field can be based on another fixed record part; and in assignment statements, that field is treated as a memory area of type CHAR regardless of the types in the fixed record part.

#### **Related concepts**

"DataItem part" on page 133  
"Non-fixed record parts"  
"Record parts" on page 135  
"Record types and properties" on page 138  
"Resource associations and file types" on page 393  
"Fixed structure" on page 27  
"Typedef" on page 28

#### **Related tasks**

"Setting the default build descriptors" on page 118  
"Setting preferences for the EGL editor" on page 118

#### **Related reference**

"Assignments" on page 456  
"EGL source format" on page 586  
"Data initialization" on page 564  
"Primitive types" on page 34

## **Non-fixed record parts**

A *non-fixed record part* defines a set of fields whose length is not necessarily known at generation time. A field in a non-fixed record can any of these:

- Dictionary
- ArrayDictionary
- An array of dictionaries or arrayDictionaries

Also, a field can be based on any of these:

- A primitive type such as STRING
- A DataItem part
- A fixed-record part
- Another record part
- An array of any of the preceding kinds

Two types of non-fixed record parts are available:

- BasicRecord, as is used for general processing but not for accessing a data store
- SQLRecord, as is used for accessing a relational database

You may use a non-fixed record in the following contexts:

- In a statement that copies data to or from a relational database
- In an assignment or move statement
- As an argument that passes data to another program or function
- As a parameter that receives data into a program or function

A record part that includes a fixed structure (with each field assigned a level number) is a fixed record part, even if the record part is of type BasicRecord or SQLRecord. For details, see *Fixed record parts*.

#### **Related concepts**

“DataItem part” on page 133  
“Fixed record parts” on page 136  
“Fixed structure” on page 27  
“Parts” on page 19  
“Record types and properties”  
“Resource associations and file types” on page 393  
“Typedef” on page 28

#### **Related tasks**

“Setting the default build descriptors” on page 118  
“Setting preferences for the EGL editor” on page 118

#### **Related reference**

“EGL source format” on page 586  
“Data initialization” on page 564  
“Primitive types” on page 34

## **Record types and properties**

Several EGL record types are available:

- “ALT\_PCBRecord”
- “BasicRecord” on page 139
- “DB\_PCBRecord” on page 139
- “DLISegment” on page 139
- “GSAM\_PCBRecord” on page 140
- “IndexedRecord” on page 141
- “IO\_PCBRecord” on page 141
- “MQRecord” on page 141
- “PSBDataRecord” on page 142
- “PSBRecord” on page 142
- “RelativeRecord” on page 142
- “SerialRecord” on page 143
- “SQLRecord” on page 143
- “VGUIRecord” on page 144

For details on what target systems support what record types, see *Record and file-type cross-reference*. For details on how record parts are initialized, see *Data initialization*.

### **ALT\_PCBRecord**

This predefined record part describes the layout of a program control block (PCB) that lets a COBOL program change the destination for an outgoing message. Rather than read or write a record of this type, you include a reference to the PCB record when you perform read or write operations in DL/I. The layout of the record is as follows:

```
Record ALT_PCBRecord
  10 terminalName char(8);
  10 * char(2);
  10 statusCode char(2);
end
```

The ALT\_PCBRecord type features a number of properties and property fields, described in *PCB record part properties*.

## BasicRecord

A basic record or fixed basic record is used for internal processing and cannot access data storage.

The part is a non-fixed record part by default; but is a fixed record part if the field definitions are preceded by level numbers.

In a fixed record part of type basicRecord, the following properties are available:

In a main program, the following program property is available:

### inputRecord

This property identifies a record that is initialized automatically, as described in *Data initialization*.

## DB\_PCBRecord

This predefined record part describes the layout of a program control block (PCB) that determines a COBOL program's access to a DL/I database. Rather than read or write a record of this type, you include a reference to the PCB record when you perform read or write operations in DL/I. The layout of the record is as follows; you can define your own record layout based on this one:

```
Record DB_PCBRecord
  10 dbName char(8);
  10 segmentLevel num(2);
  10 statusCode char(2);
  10 procOptions char(4)
  10 * char(4);
  10 segmentName char(8);
  10 keyAreaLen int;
  10 numSensitiveSegs int;
  10 keyArea char(32767);
end
```

The DB\_PCBRecord type features a number of properties and property fields, described in *PCB record part properties*.

## DLISegment

This record type may be either fixed length or non-fixed length. If it is non-fixed length, you must specify the lengthItem property (explained later in this section) in a set-value block.

The DLISegment record type holds the data you will read or write from a DL/I database segment. You can read a segment by invoking one of several possible **get** statements; you can write a segment by invoking an **add** or **replace** statement; and you can remove a segment from the file by invoking a **delete** statement.

Fields within the record must exactly match the fields in the database segment in length and type. For purposes of clarity or to aid migration, the names of the fields in the record definition do not need to match the field names in the DL/I segment. If any field name in the record does not match, however, you must use a set-value

block to specify the eight-character `dliFieldName` property for that field. All names will be folded (converted to upper case) at generation time.

Use a set-value block to specify any or all of the following properties of the `DLISegment` record type:

**segmentName char(8)**

The name of the runtime DL/I segment. Complete this field if the name of your `DLISegment` record does not match the name of the related DL/I segment.

**hostVarQualifier String**

If you specify a host variable qualifier name, the code generator will use that name rather than the record name to qualify keyItems referenced in the default qualified segment search argument (SSA). If your DL/I segment record name does not match a program variable identifier, make sure this field does.

**lengthItem FieldReference**

If you use variable length segments in your DL/I database, you must, at run time, store the total length of your `DLISegment` record within that record.

**lengthItem** contains the name of the field where you store that record length.

**keyItem FieldReference**

If you have an index or sequence key field for the record, specify the name of that field here.

In addition, you can define the following property for any of the fields in a `DLISegment` record:

**dliFieldName char(8)**

If you assigned a name to a fields in your `DLISegment` record that does not match the name of the equivalent field in the DL/I database segment, you must provide the name of that equivalent field here. Case is not an issue; all field names will be folded (converted to upper case) during generation.

The following example shows a set value block that uses these properties:

```
Record CustomerRecord type DLISegment {
  segmentName = "STSCCST",
  hostVarQualifier = "STSCCST",
  lengthItem = "mySegementLength",
  keyItem = "customerNo" }

10 customerNo char(6)      { dliFieldName = "STQCCNO" };
10 mySegementLength int;
...
end
```

**GSAM\_PCBRecord**

This predefined record part describes the layout of a program control block (PCB) that determines a COBOL program's access to a simple sequential data set such as a tape file or SYSIN. Rather than read or write a record of this type, you include a reference to the PCB record when you perform read or write operations in DL/I. The layout of the record is as follows:

```
Record GSAM_PCBRecord
10 dbName char(8);
10 * num(2);
10 statusCode char(2);
10 procOptions char(4)
```

```

10 * char(20);
10 recordSearchArg bigint;
10 undefinedRecordLen int;
end

```

The GSAM\_PCBRecord type features a number of properties and property fields, described in *PCB record part properties*.

## IndexedRecord

An indexed record is a fixed record that lets you to work with a file that is accessed by a *key value*, which identifies the logical position of a record in the file. You can read the file by invoking a **get**, **get next**, or **get previous** statement. Also, you can write to the file by invoking an **add** or **replace** statement; and you can remove a record from the file by invoking a **delete** statement.

The properties of a part of type indexedRecord include these:

### fileName

This property is required. For details on the meaning of your input, see *Resource associations (overview)*. For details on the valid characters, see *Naming conventions*.

### keyItem

This required property can only be a structure field that is unique in the same record. You must use an unqualified reference to specify the key field; for example, use *myItem* rather than *myRecord.myItem*. (In an EGL statement, however, you can reference the key field as you would reference any field.)

See also *Properties that support variable-length records*.

## IO\_PCBRecord

This predefined record part describes the layout of a program control block (PCB) that lets a COBOL program communicate with a user through a terminal. Rather than read or write a record of this type, you include a reference to the PCB record when you perform read or write operations in DL/I. The layout of the record is as follows:

```

Record IO_PCBRecord
10 terminalName char(8);
10 * char(2);
10 statusCode char(2);
10 * char(8);
12 localDate decimal(7);
12 localTime decimal(7);
10 inputMsgSegNum int;
10 userid char(8);
10 groupName char(8);
10 * char(12);
12 currentDate decimal(7);
12 currentTime decimal(11);
12 utcOffset hex(4);
10 userIdIndicator char(1);
end

```

The IO\_PCBRecord type features a number of properties and property fields, described in *PCB record part properties*.

## MQRecord

An MQ record is a fixed record that lets you access an MQSeries message queue. For details, see *MQSeries support*.



## PSBDataRecord

As used for DL/I processing or on IMS, a record of type PSBDataRecord is a fixed record that is organized as follows:

```
Record PSBDataRecord
  psbName char(8);
  psbRef int;
end
```

You can use the record to interact with the system variable **DLILib.psbData**, which contains both the name of the runtime PSB and an address with which that PSB is accessed. The record is also useful if you need to pass the PSB (really, a name and address) to another program or to receive the PSB from another program.

## PSBRecord

This record part defines the structure of the runtime program specification block (PSB) and includes a series of PCB records. When you develop a PCB record for a database, for example (record type DB\_PCBRecord), you represent a segment hierarchy by assigning values to the complex property **@PCB**. As shown in a later example, that property identifies the PCB name and type and includes a **hierarchy** field, which in turn contains a sequence of **@Relationship** properties.

If you use the CBLTDLI interface, you must declare the following PCBs first, in order, to provide the proper offset for the "access by index" that CBLTDLI requires:

1. iopcb for an IO PCB
2. ELAALT for an alternate index PCB
3. ELAEXP for an alternate express PCB

The following is an example PSB record for a customer database:

```
Record CustomerPSB type PSBRecord { defaultPSBName="STBICLG" }
// three PCBs required for call interface CBLTDLI
iopcb IO_PCBRecord { @PCB { pcbType = TP } };
ELAALT ALT_PCBRecord { @PCB { pcbType = TP } };
ELAEXP ALT_PCBRecord { @PCB { pcbType = TP } };

// database PCB
customerPCB DB_PCBRecord { @PCB {
  pcbType = DB,
  pcbName = "STDCDBL",
  hierarchy = [
    @Relationship { segmentRecord = "CustomerRecord" },
    @Relationship {
      segmentRecord = "LocationRecord", parentRecord = "CustomerRecord" },
    @Relationship {
      segmentRecord = "CreditRecord", parentRecord = "CustomerRecord" },
    @Relationship {
      segmentRecord = "HistoryRecord", parentRecord = "CustomerRecord" },
    @Relationship {
      segmentRecord = "OrderRecord", parentRecord = "LocationRecord" },
    @Relationship {
      segmentRecord = "ItemRecord", parentRecord = "OrderRecord" } ] };
end
```

For more information on the properties of a PCB record, see *PCB record type properties*.

## RelativeRecord

A relative record is a fixed record that lets you work with a data set whose records have these properties:

- The records are fixed length.

- The records can be accessed by an integer that represents the sequential position of the record in the file.

The properties of a part of type `relativeRecord` are as follows:

#### **fileName**

The contents of this field depend on context. For details, see *Resource associations and file types*. For details on the valid characters, see *Naming conventions*. This is a required field.

#### **keyItem**

The key field can refer to any of these areas of memory:

- A structure field in the same record
- A structure field in a record that is global to the program or is local to the function that accesses the record
- A primitive variable that is global to the program or is local to the function that accesses the record

You must use an unqualified reference to specify the key field. For example, use `myItem` rather than `myRecord.myItem`. (In an EGL statement, you can reference the key field as you would reference any field.) The key field must be unique in the local scope of the function that accesses the record or must be absent from local scope and unique in global scope.

The key field has these characteristics:

- Has a primitive type of BIN, DECIMAL, INT, or NUM
- Contains no decimal places
- Allows for 9 digits at most

Only the **get** and **add** statements use the key field, but the key field must be available to any function that uses the record for file access.

### **SerialRecord**

A serial record is a fixed record that lets you access a sequentially accessed file or data set. You can read from the file by invoking a **get** statement, and a series of **get next** statements reads the file records sequentially, from the first to the last. You can write to the file by invoking an **add** statement, which places a new record at the end of the file.

Serial record properties include the following:

#### **fileName**

The contents of this field depend on context. For details, see *Resource associations and file types*. For details on the valid characters, see *Naming conventions*. This is a required field.

See also *Properties that support variable-length records*.

### **SQLRecord**

An SQL record is a record that provides special services when you access a relational database.

The part is a non-fixed record part by default; but is a fixed record part if the field definitions are preceded with level numbers.

Each part has the following optional properties:

**tableNames**

An entry in **tableNames** identifies an SQL table associated with the part. You may reference multiple tables in a join, but restrictions ensure that you do not write to multiple tables with a single EGL statement. You may associate a given table name with a *label*, which is an optional, short name used to reference the table in an SQL statement.

**defaultSelectCondition**

The property specifies conditions that become part of the WHERE clause in default SQL statements. The WHERE clause is meaningful when an SQL record is used in an EGL **open** or **get** statement or in statements like **get next** or **get previous**.

In most cases, the SQL default select condition supplements a second condition, which is based on an association between the key-field values in the SQL record and the key columns of the SQL table.

**tableNameVariables**

You can specify one or more variables whose content at run time determines what database tables to access, as described in *Dynamic SQL*.

**keyItems**

Each key field can only be a structure field that is unique in the same record. You must use an unqualified reference to specify each of those fields; for example, use *myItem* rather than *myRecord.myItem*. (In an EGL statement, however, you can reference a key field as you would reference any field.)

For details, see *SQL support*.

**VGUIRecord**

A VGUIRecord part is a generatable part and is the basis of a VGUI record, which is a VGWebTransaction program or function variable that makes communication possible between the program and a specific Web page.

The properties of a part of type VGUIRecord include these:

**validatorFunction**

Specifies the EGL function validator function, which is invoked after all the field validators are invoked, as described in *Validation in Web applications built with EGL*

**runValidatorFromProgram**

Specifies whether the validator function is in the VGUI record bean or is in the program that receives data from the VGUI record bean.

**commandValueItem**

Identifies the VGUI record field that contains the value of the SUBMIT button clicked by the user.

**Related concepts**

“DL/I database support” on page 310

“Dynamic SQL” on page 288

“Fixed record parts” on page 136

“MQSeries support” on page 336

“Non-fixed record parts” on page 137

“Record parts” on page 135

“Resource associations and file types” on page 393

“SQL support” on page 277

### Related reference

"add" on page 661

"close" on page 669

"Data initialization" on page 564

"delete" on page 673

"execute" on page 677

"get" on page 687

"get next" on page 701

"get previous" on page 708

"MQ record properties" on page 772

"Naming conventions" on page 778

"open" on page 722

"PCB record part properties"

"prepare" on page 736

"Properties that support variable-length records" on page 860

"Record and file type cross-reference" on page 860

"replace" on page 738

"SQL item properties" on page 68

"terminalID" on page 1075

## PCB record part properties

EGL offers the following program communication block (PCB) record parts:

### IO\_PCBRecord

PCB of pcbType=TP that lets a COBOL telecommunication program communicate with a user through a terminal.

### ALT\_PCBRecord

PCB of pcbType=TP that lets a COBOL telecommunication program change the destination for an outgoing message.

### DB\_PCBRecord

PCB of pcbType=DB that determines a COBOL program's access to a DL/I database.

### GSAM\_PCBRecord

PCB of pcbType=GSAM that determines a COBOL program's access to a simple sequential data set such as a tape file or SYSIN.

For more about these record parts, including their definitions and fields, see *Record types and properties*.

All four record parts share the following properties:

### @PCB

This complex property defines basic characteristics of the runtime PCB. The property is required for all four record parts, unless the record is a redefinition of a previous PCB. Use a set-value block to specify any or all of the following fields for the @PCB complex property:

### pcbType PCBKind

This field tells EGL what kind of PCB the record defines. The **pcbType** should be one of the following values of the PCBKind enumeration:

#### DB

A database PCB (DB\_PCBRecord)

**GSAM**

A sequential file PCB (GSAM\_PCBRecord)

**TP**

A telecommunication program PCB (IO\_PCBRecord or ALT\_PCBRecord)

The **pcbType** is required if you plan to use the PSB in both the IMS and CICS environments. At run time in the CICS environment, EGL checks the value of **pcbType** and ignores GSAM or TP PCBs.

**pcbName char(8)**

If you plan to use the AIBTDLI interface, you must tell EGL the names of the PCBs in your DL/I PSB. You can do this in two ways:

- You can give your PCB records the same names that they have in the DL/I PSB.
- You can use the **pcbName** field to associate the DL/I PCB name with your EGL PCB name.

If you did not give your PCB record the same name as the runtime PCB and you use the AIBTDLI interface, you can specify the name of the runtime PCB here. The CBLTDLI interface does not use this name, so if you use CBLTDLI, the name of your EGL PCB record must match the name of the runtime PCB.

**secondaryIndex char(8)**

If you use a secondary index to order database records, this is the name of the index in the DL/I PCB. This name is used on the left side of a default qualified SSA for a root segment instead of the key item specified for the root segment record.

**secondaryIndexItem FieldReference**

If you use a secondary index, this is the name of the field in the associated PCB record that holds the secondary index value. This is the host variable value in the default qualified SSA of the root segment for the given PCB.

**hierarchy**

This field contains an array of **@Relationship** properties that describe the relationships between the database segments. The structure of those properties is as follows:

**@Relationship**

Each **@Relationship** property introduces a set-value block with the following fields:

**segmentRecord STRING**

The name of a segment in the hierarchy

**parentRecord STRING**

The name of the parent record for the segmentRecord. If the segmentRecord specifies the root segment, you can either set parentRecord to 0 or omit the parentRecord field.

If your target environment is IMS using the CBLTDLI interface, you must define the following PCBs, in order, by name. EGL uses these PCBs for internal operations:

**iopcb**

A variable of type IO\_PCBRecord, representing a PCB of PCBType TP.

**elaalt**

A variable of type ALT\_PCBRecord, representing an Alternate Index PCB of PCBType TP.

**elaexp**

A variable of type ALT\_PCBRecord, representing an Alternate Express PCB of PCBType TP.

If necessary, use the **pcbName** field above to associate these PCBs with their respective DL/I PCB names.

The following example shows :

```
Record CustomerPSB type PSBRecord { defaultPSBName="STBICLG" }
// three PCBs required for CBLTDLI on IMS
iopcb IO_PCBRecord { @PCB { pcbType = TP } };
elaalt ALT_PCBRecord { @PCB { pcbType = TP } };
elaexp ALT_PCBRecord { @PCB { pcbType = TP } };

// database PCB defining three segment levels
customerPCB DB_PCBRecord { @PCB {
  pcbType = DB,
  pcbName = "STDCDBL",
  hierarchy = [
    @Relationship { segmentRecord = "myCustomerRecordPart" },
    @Relationship {
      segmentRecord = "myLocationRecordPart", parentRecord = "myCustomerRecordPart" },
    @Relationship {
      segmentRecord = "myOrderRecordPart", parentRecord = "myLocationRecordPart" },
    @Relationship {
      segmentRecord = "myItemRecordPart", parentRecord = "myOrderRecordPart" },
    @Relationship {
      segmentRecord = "myCreditRecordPart", parentRecord = "myCustomerRecordPart" },
    @Relationship {
      segmentRecord = "myHistoryRecordPart", parentRecord = "myCustomerRecordPart" }]]];
```

**Related reference**

“Record types and properties” on page 138

---

## Creating an EGL program part

An EGL program part is the main logical unit used to generate a Java program, a Java wrapper, or an Enterprise JavaBean session bean. For more information, see *Program part*.

A program part is automatically added to an EGL source file and named appropriately when you create the file in the workbench. Source file specifications allow only one program part per file, and require a program name that matches the file name.

To create a program file with a program part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Program**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the program name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myEGLprg. Select an EGL program type (for details, see *Basic program in EGL source format* or *TextUI program in EGL source format* , or *VGWebTransaction program in EGL source format*). If the program

part is a main program, click to remove the check mark from Create as called program. A VGWebTransaction program cannot be called from another program.

4. Click the **Finish** button.

#### Related concepts

"EGL projects, packages, and files" on page 15

"Introduction to EGL" on page 1

"Program part"

#### Related tasks

"Creating an EGL source folder" on page 129

#### Related reference

"Basic program in EGL source format" on page 842

"Creating an EGL source file" on page 130

"Naming conventions" on page 778

"Text UI program in EGL source format" on page 844

"VGWebTransaction program in EGL source format" on page 847

## Program part

A *program part* defines the central logical unit in a runtime Java program. For an overview of main and called programs and of the program types (basic, textUI, and VGWebTransaction), see *Parts*.

Any kind of program part includes a function called *main*, which represents the logic that runs at program start up. A program can include other functions and can access functions that are outside of the program. The function *main* can invoke those other functions, and any function can give control to other programs.

The most important program properties are as follows:

- Each *parameter* references an area of memory that contains data received from a caller. Parameters are global to the program and are valid only in called programs.
- Each *variable* references an area of memory that is allocated in and global to the program.
- A *form group* is a collection of forms that present data to the user:
  - A basic or VGWebTransaction program can present data to a printer by way of *print forms*
  - A textUI program can present data interactively (by way of *text forms*) or to a printer

For details, see *FormGroup part*.

- An *input record* is an area of global memory that receives data when control is transferred asynchronously from another program. An input record is available only in a main program.
- In main textUI programs, the *segmented* property determines what actions are taken automatically before the program issues a **converse** statement to present a text form. For details, see *Segmentation*.
- Also in text programs, an *input form* has one of two purposes at program start up:
  - The form is presented to a user who invokes the program from a monitor or terminal



- Alternatively, data that was entered by a user is received into the input form, which is a memory area in the program itself. This situation applies only in the case of a *deferred program switch*, which is a two-step transfer of control that is caused by a variant of the **show** statement--
  1. A program submits a text form to the user, then terminates
  2. The user submits the form, and by virtue of information in the form, the submission automatically invokes a second program, which contains the input form
- In VGWebTransaction programs, an *input UI record* is used during a deferred program switch. In this case, the two-step process is caused by a variant of the **show** statement and works as follows:
  1. A program submits a Web page to the user, then terminates.
  2. The user submits a form from the Web page, and by virtue of information available to the EGL runtime, the submission automatically invokes a second program, which contains data from the Web page. That data is received into the program's input UI record. (Also, data directly from the originating program is received into the receiving program's input record.)

For a complete list of program properties, see *Program part properties*.

#### **Related concepts**

"FormGroup part" on page 183  
 "Function part" on page 150  
 "Parts" on page 19  
 "References to variables in EGL" on page 59  
 "Segmentation in text applications" on page 189

#### **Related tasks**

"Creating an EGL program part" on page 147

#### **Related reference**

"Content assist in EGL" on page 577  
 "Data initialization" on page 564  
 "EGL source format" on page 586  
 "EGL statements" on page 88  
 "Program part in EGL source format" on page 841  
 "Program part properties" on page 856

---

## **Creating an EGL function part**

A function part is a logical unit that either contains the first code in a program or is invoked from another function. EGL function parts are contained in EGL files. To create an EGL function part, do as follows:

1. Identify an EGL file to contain the function part and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the function part according to EGL syntax (for details, see *Function part in EGL source format*). You can use content assist to place an outline of the function part syntax in the file.
3. Save the EGL file.

#### **Related concepts**

"EGL projects, packages, and files" on page 15  
 "Function part" on page 150



### Related tasks

“Creating an EGL source file” on page 130

“Using the EGL templates with content assist” on page 131

### Related reference

“Content assist in EGL” on page 577

“Function invocations” on page 613

“Function part in EGL source format” on page 621

“Naming conventions” on page 778

## Function part

A *function part* is a logical unit that either contains the first code in the program or is invoked from another function. The function that contains the first code in the program is called *main*.

The function part can include the following properties:

- A *return value*, which describes the data that the function part returns to the caller
- A set of *parameters*, each of which references memory that is allocated and passed by another logic part
- A set of other *variables*, each of which allocates other memory that is local to the function
- EGL statements
- A specification as to whether the function requires the program context, as described in *containerContextDependent*

The function *main* is unusual in that it cannot return a value or include parameters, and it must be declared inside a program part.

### Related concepts

“Parts” on page 19

“Program part” on page 148

“References to variables in EGL” on page 59

“SQL support” on page 277

### Related reference

“containerContextDependent” on page 557

“Data initialization” on page 564

“EGL source format” on page 586

“Function invocations” on page 613

“Function part in EGL source format” on page 621

“EGL statements” on page 88

---

## Creating an EGL Interface part

To create an EGL source file that contains an EGL interface part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Interface**. The New EGL Interface Part wizard opens
3. In the **Source folder** field, select a source folder to hold the new file.
4. In the **Package** field, select a package to hold the new file.
5. In the **EGL source file name** field, type a name for the new file.

6. Under **EGL Interface Type** select a type of interface to create. For more information, see *Interface part*.
7. If you have selected **Java Object** as the type of interface part to create, you can select existing Java Object interface parts for the new interface to extend. To select interfaces for the new interface to extend, follow these steps:
  - a. Click **Add**. The Extended Interfaces Selection window opens.
  - b. In the **Choose interfaces** field, type a search string.
    - A question mark (?) represents any one character
    - An asterisk (\*) represents a series of any characters
 The interface parts that match the search string are listed under **Matching parts**.
  - c. Click the interface part you want the new interface part to extend.
  - d. Click **Add**.
  - e. When you are finished choosing interface parts, click **OK**.
8. Click **Finish**.

You can also create an interface part based on the functions in a service part. See *Creating an Interface part from a Service part*.

#### Related concepts

"EGL interfaces"

"EGL projects, packages, and files" on page 15

"Introduction to EGL" on page 1

"EGL services and Web services" on page 158

#### Related tasks

"Creating an Interface part from a Service part" on page 154

"Creating an EGL source folder" on page 129

#### Related reference

"Interfaces of type BasicInterface" on page 636

"Interfaces of type JavaObject" on page 637

"Interface part in EGL source format" on page 633

"Service part in EGL source format" on page 869

## EGL interfaces

EGL interfaces let you access an EGL service, a Web service (which may be written in EGL), or Java code.

The interface is an EGL part that includes a set of function descriptions. As shown in the following example, each function description has an ending semicolon (;) and includes only a function name, parameters, and return type:

```
Interface HelloWorld type JavaObject
{javaName = "HelloWorld",
  packageName = "com.ibm.examples.helloWorld"}

  function sayHello(name String)
    returns (String);
end
```

Instead of using an Interface part directly in your code, you create a variable that is based on that part. At run time, the variable refers to one of these:

- A service that runs at a specific location; or

- Java code--
  - A Java object;
  - A Java class that has a static method; or
  - A Java interface.

You can use interfaces in these ways:

#### For services

You can use an interface of type `BasicInterface` to define the purpose of a service not yet written or to provide access to an existing service--

- **You can use interfaces to help you architect your organization's services.**

You can create an interface that describes the functionality that you want to have coded in an EGL service. After the interface is complete, you or others can code the service, which is said to *implement the interface*. The primary meaning is that the service contains every function described in the interface. The interface provides a kind of contract that the service must fulfill.

This use of interfaces provides the following benefits:

- Helps people in your organization to think clearly about what operations are needed in a service before service development begins
- Allows developers to finish client applications while the service code is under development, as is possible when the client code interacts with variables that are based on Interface parts, not Service parts

A service can implement multiple interfaces, and multiple services can implement the same interface.

- **Interfaces give developers access to your organization's services while shielding them from knowledge of the implementation.**

The runtime effect of declaring a variable of an Interface part is the same as the effect of declaring a variable of the related Service part; but use of the interface lets you avoid disclosing the logic inside the service. Keeping the logic away from others might be appropriate for competitive reasons (you may want to hide the source code from developers outside of your organization) or to reduce complexity (you may want developers -- whether in or out of your organization -- to focus on the functionality that the service provides rather than on the details of the implementation).

- **Interfaces let you access a Web service regardless of the language or location of the service implementation.**

Access of a Web service created outside of your organization requires that you have the service-specific Web Service Description Language (WSDL) definition, which details how to access the service.

You use the WSDL definition as input to the EGL WSDL wizard, which creates a service-specific Interface part that will be the basis of a variable in your code.

When you are accessing a service by way of an interface, the interface must be bound to the location where the service implementation resides. This binding includes details on the protocol used to access the implementation.

The usual situation is that you specify the binding information in a variable that is declared in a special type of EGL library called a *service binding library*, which is described in *Library part of type ServiceBindingLibrary*. Another possibility is that an interface receives the binding in an assignment statement, as is possible when the interface or service on the right side of the assignment statement is declared in a services binding library or when that interface or

service received a binding in a previous assignment statement. A third possibility is that an interface that is acting as a function parameter receives the binding by receiving an argument that already has a binding.

### **For Java code**

You can use an EGL interface of type `JavaObject` to access some or all of the functionality that is available in a Java interface or class. In this usage, an EGL interface provides a view of the Java code.

The following benefits come from using EGL interfaces in this way:

- Coding and maintenance of the EGL logic is easier because the complexity of the Java code is largely hidden in the underlying Java implementation
- The number of programmers who can access Java code from EGL is maximized

An efficient strategy for your organization is to ask a developer who has expertise in both EGL and Java to create Interface parts for use by other EGL developers. For a list of interface parts that are provided for general use, see *Interfaces of type JavaObject*.

EGL provides a large set of interfaces to allow Web developers to access JSF user-interface controls at run time; for example, to change the color of a text box in response to a user's input. For an overview of that capability, see *JSF component tree*.

When you interact with the JSF component tree, you can ignore the fact that (as of EGL version 6.0.1), EGL interface technology does not allow you to instantiate a Java object from within your EGL code. For details on the issue, see *Instantiation and EGL interfaces of type JavaObject*.

EGL interfaces supplement two other Java-specific capabilities in EGL:

### **Java access functions**

The Java access functions are EGL system functions that allow your generated Java code to access native Java objects and classes; specifically, to access the public methods, constructors, and fields of the native code.

### **EGL Java wrapper**

An EGL Java wrapper is a set of Java classes that are generated for you and that act as an interface between the following executables:

- A servlet or a hand-written Java program, on the one hand
- A EGL-generated program or EJB session bean, on the other

### **Related concepts**

"Instantiation and EGL interfaces of type `JavaObject`" on page 155

"JSF component tree" on page 229

"Library part of type `ServiceBindingLibrary`" on page 172

### **Related tasks**

"Creating an EGL Interface part" on page 150

"Creating an EGL Service part" on page 157

"Creating an Interface part from a Service part" on page 154

### **Related reference**

"Best practices for services and related interfaces in EGL" on page 162

“EGL library ServiceLib” on page 986

“Interface part in EGL source format” on page 633

## Creating an Interface part from a Service part

You can use a service part as a model for an interface part. Follow these steps to create an Interface part that references one or more of the functions in an existing service part:

1. In the Project Explorer view, right-click on an EGL source file that includes an EGL Service part and then click **Extract EGL Interface**. The New EGL Part window opens.
2. In the **Source folder** field, select a source folder to hold the new file.
3. In the **Package** field, select a package to hold the new file.
4. In the **EGL source file name** field, type a name for the new file that will contain the new Interface part.
5. In the **Functions** field, select the functions which you want to include in the new Interface part.  
You can use the **Select All** and **Deselect All** buttons to select or deselect the functions.
6. If you want to overwrite an EGL source file, select the **Overwrite existing files** check box.
7. Click **Finish**.

### Related concepts

“EGL interfaces” on page 151

“EGL services and Web services” on page 158

### Related tasks

“Creating an EGL Interface part” on page 150

### Related reference

“Best practices for services and related interfaces in EGL” on page 162

“Interface part in EGL source format” on page 633

“Interfaces of type BasicInterface” on page 636

## Creating EGL Parts from a WSDL file

You can create a group of EGL parts based on a WSDL file to use when accessing a Web service. Following is a list of EGL parts created from the elements in the WSDL file:

- DataItem parts are created from the wsdl:types elements.
- Interface parts are created from the wsdl:porttype elements.
- Abstract functions are created from the wsdl:operation elements.

Also, if the WSDL file is in an EGL Web project, not an EGL project, the following EGL parts are created in addition to the parts listed above:

- A service binding library is created from the wsdl:service elements.
- Interface variables are created in the service binding library from the wsdl:port elements.

Follow these steps to create EGL parts from a WSDL file:

1. In the Project Explorer view, right-click the WSDL file and then click **Create EGL Interfaces and Binding Library** for an EGL Web project or **Create EGL Interfaces** for an EGL project. The New EGL Part window opens.

2. Select the interfaces from the file to be created by selecting the check boxes next to the names of the interfaces.
3. To create Web service binding libraries from the WSDL file, select the **Create Web Services Bindings** check box. This option is available only if the WSDL file is in an EGL Web project.
4. To overwrite files with the same name as the new files being created, select the **Update existing files** check box.
5. Click **Next**.
6. On the New EGL Interface page, identify a source folder, package, and source file name for the new interface parts.
7. Each interface you selected on the previous page is represented by a tab on this page. For each interface, enter a name in the **Interface Name** field. This name will be the name of the EGL interface part.
8. For each interface, select the functions you want to include in the EGL interface part by selecting or clearing the check box next to the function name.
9. If you selected **Create Web Services Bindings**, click **Next**. Otherwise, click **Finish**.
10. On the New EGL Service Binding Library page, identify a source folder, package, and source file name for the new service binding library.
11. Select the service binding variables to be created by selecting or clearing the check box next to each variable.
12. Click **Finish**.

#### Related concepts

"DataItem part" on page 133  
 "EGL interfaces" on page 151  
 "EGL services and Web services" on page 158  
 "Library part of type ServiceBindingLibrary" on page 172

#### Related tasks

"Creating an EGL Interface part" on page 150

## Instantiation and EGL interfaces of type `JavaObject`

When you use an interface of type `JavaObject`, you often need to access a static method that creates an object at run time. This requirement is in effect because the EGL interface technology does not allow you to instantiate a Java object from within your EGL code.

Consider the following EGL source file:

```
package myInterfaces;

Interface HelloWorld type JavaObject
{javaName = "HelloWorld",
  packageName = "com.ibm.examples.helloWorld"}

function sayHello(name String)
  returns (String);
End
```

The related Java class would need to be available at both compile time and run time:

```
package com.ibm.examples.helloWorld;

class HelloWorld
```

```

{
    // Java class constructor
    public HelloWorld()
    {
    }

    // implementation of the function
    // that is referenced in the interface
    public String sayHello(String name)
    {
        return "Hello " + name;
    }
}

```

An additional EGL Interface part is appropriate, however:

```

package myInterfaces;

Interface HelloWorldFactory
{
    javaName = "HelloWorldFactory",
    packageName = "com.ibm.examples.helloWorld"
    static function createHelloWorld()
        returns (HelloWorld);
}
End

```

The Interface part HelloWorldFactory refers to the following Java class, which includes a static method that returns an object of type HelloWorld:

```

package com.ibm.examples.helloWorld;

class HelloWorldFactory
{
    // Java class constructor
    public HelloWorldFactory()
    {
    }

    // implementation of the function
    // that is referenced in the interface
    public static HelloWorld createHelloWorld()
    {
        return new HelloWorld;
    }
}

```

An EGL program might include the following code:

```

package myDriver;
import myInterfaces.*;

program myProgram { }

function myFunction()
    echo STRING;

    // create a variable of type HelloWorld;
    // this reference variable points to NIL
    hwVar HelloWorld;

    // assign a Java object to the variable
    hwVar = HelloWorldFactory.createHelloWorld();

    // access any Java method that is in
    // the object of type HelloWorld
    // and has a corresponding
    // function description

```

```

// in the EGL interface part
echo = hwVar.sayHello("John Doe");
End
End

```

When you are writing a pageHandler that interacts with the JSF component tree, you create variables that are based on EGL interface parts; however, you are not concerned with object creation because EGL handles the issue at run time. For an overview, see *JSF component tree*.

#### Related concepts

“EGL interfaces” on page 151

“JSF component tree” on page 229

#### Related tasks

“Creating an EGL Interface part” on page 150

#### Related reference

“Interface part in EGL source format” on page 633

“Interfaces of type JavaObject” on page 637

---

## Creating an EGL Service part

To create an EGL source file that contains an EGL service part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Service**. The New EGL Service Part wizard opens
3. In the **Source folder** field, select a source folder to hold the new file.
4. In the **Package** field, select a package to hold the new file.
5. In the **EGL source file name** field, type a name for the new file.

Since the name of the new service will be identical to the name of the file, type a name that adheres to EGL part name conventions.

6. If you want the new service part to implement any EGL interface parts, follow these steps:
  - a. Click **Add**. The Implemented Interfaces Selection window opens.
  - b. In the **Choose interfaces** field, type a search string.
    - A question mark (?) represents any one character
    - An asterisk (\*) represents a series of any characters
 The interface parts that match the search string are listed under **Matching parts**.
  - c. Click the interface part you want the service part to implement.
  - d. Click **Add**.
  - e. When you are finished choosing interface parts, click **OK**.

The interface parts you chose are listed in the **Implements Interfaces** field. You can remove an interface part by clicking it and then clicking **Remove**.

7. New service parts can include functions that call an EGL called program. If you want the new service part to have functions that call an EGL called program, follow these steps:
  - a. Click **Show Advanced**.
  - b. Click the **Add** button next to **Basic Called Programs**. The Basic Called Program Selection window opens.



- c. In the **Choose Programs** field, type a search string.
  - A question mark (?) represents any one character
  - An asterisk (\*) represents a series of any characters

The programs that match the search string are listed under **Matching parts**.

- d. Click the program name.
- e. Click **Add**.
- f. When you are finished choosing programs, click **OK**.

The programs you chose are listed in the **Basic Called Programs** field. You can remove an interface part by clicking it and then clicking **Remove**. For each of the programs in this field, the new service part will have a function that calls the program.

- 8. Click **Finish**.

#### Related concepts

"EGL projects, packages, and files" on page 15

"Introduction to EGL" on page 1

"EGL services and Web services"

"EGL interfaces" on page 151

#### Related tasks

"Creating an EGL source folder" on page 129

#### Related reference

"Naming conventions" on page 778

"Interface part in EGL source format" on page 633

"Service part in EGL source format" on page 869

## EGL services and Web services

A *service* is a set of operations that can be invoked by a *client*, which is an application component that resides on the same or another platform. A service that you develop in EGL can be deployed in either or both of these ways:

- As an *EGL service*, which can be accessed from EGL code either directly or by way of a TCP/IP connection. The client can be a program, handler, library, or other service.
- As a *Web service*, which can be accessed from code that runs in a Web application server. The client can be a JSP, servlet, or Java client application; or a script or an executable program written in any of several languages such as C++, Perl, Visual Basic, or JavaScript<sup>™</sup>. Access is by way of an HTTP connection.

**Note:** EGL 6.0.1 supports deployment of Web services and Web service clients only on WebSphere Application Server versions 5.1.x and 6.0. Also, EGL 6.0.1 supports deployment of Web services and Web service clients only when WebSphere Application Server is using the IBM WebSphere runtime environment, instead of the IBM SOAP or Apache Axis runtime environments. For more information, refer to the documentation for WebSphere Application Server.

At development time, the service is a part that includes implementation code. Here is a simple example:

```
Service echoString
function returnString
  (inputString string in)
  returns (string)
```

```

        return (inputString);
    end
end

```

The Service part is similar to a library part:

- Neither can include a user interface
- Either can include variables (and constants) that are declared as private
- Either can include functions that are declared as private or public

Libraries and services differ in the following ways:

- The code that uses the service can access only public functions, not constants or variables, whereas the code that uses a library can access public constants and variables
- Any global memory in the service is initialized at each invocation of the service, whereas changes to the global memory in a library persist for the life of the run unit

Instead of using a Service part directly in your code, you create a variable that is based on that part. The Service part includes the logic for the runtime service but does not include the details necessary for access. At run time, however, the variable *is bound* (refers) to a service that runs at a specific location. The binding includes details on the protocol that is used to access the service.

You specify the binding information in a variable declared in a *service binding library*, which is a type of EGL library. The variable is global to the run unit, as is true of any variable accessed directly from an EGL library. For most uses, the initial binding for that variable provides all the service-specific functionality needed by any client in the run unit.

The setting of the library property **runtimeBind** determines whether the initial binding of a variable to a service is based on a decision made at development time or at deployment time:

- If you set the property to *no* (as is the default), the decision is made at development time. The binding that you detail in the library is in effect when the run unit starts.
- If you set the property to *yes*, the decision is made at deployment time. The binding information is retrieved at the start of the run unit, from a property file that is specific to the library.

For advanced uses, EGL provides these capabilities:

- You can get and set the service location at run time by invoking the functions described in *ServiceLib*
- You can use multiple variables to simulate dynamic binding, as described in “Service bindings at run time” on page 160

To provide access to a non-EGL Web service, you must use an EGL interface. An interface also provides a secondary way to access a service that was developed in EGL. For an introduction to this subject, see *EGL interfaces*.

## Runtime processes

Any EGL client accesses a service by way of a generated proxy that contains function descriptions equivalent to those in the service. The following configurations are possible:

- An EGL client accesses an EGL service by direct connection, in which case the client and service are in the same run unit.
- An EGL client accesses an EGL service by way of TCP/IP, in which case the EGL runtime is used on both the client and service side of the interaction
- An EGL client accesses a Web service, in which case a Web Service Description Language (WSDL) definition is used on both sides of the transmission:
  1. The generated proxy invokes a Java JAX-RPC stub that was created when EGL generated the service binding library.
  2. The stub uses the information from the WSDL definition to send an XML-based, Simple Object Access Protocol (SOAP) message to the service. That message includes a valid mapping between an EGL-generated data type and the type of data submitted to or received from the service.

**Note:** In most cases the default transformations are sufficient; but you can override a default by setting the @xsd property for a particular field, as described in @xsd.

3. The stub receives the returned SOAP message, which is either business data from the Web service or an exception of type ServiceInvocationException. For details on the latter, see *EGL system exceptions*.
4. The stub uses the information from the WSDL definition, in this case to convert data from the SOAP data types into the Java data types expected by the proxy.
5. The stub returns data and control to the proxy.
6. The proxy returns control to the client.

When an EGL-generated Web service is in use, the following events occur on the service side of transmission:

- In response to its invocation, the Web services runtime code uses the service-side WSDL definition to create data that the service can use.

**Note:** EGL generates a WSDL definition with the Web service and uses (in most cases) a default transformation of EGL-generated fields to WSDL elements. You can override a default by setting the @xsd property for a particular field, as described in @xsd. In this case, the purpose of setting the property is to establish a set of validation rules that limit the kind of data that will be provided to the service at run time.

- The Web service runtime code submits the data to the Web service and acts as follows when the Web service returns data:
  1. Uses the service-side WSDL definition to create an XML-based SOAP message
  2. Returns that message to the client

## Service bindings at run time

Consider the case of two services that provide identical operations but are on different servers. You can arrange for the client to select one or another service in response to a runtime condition such as a value retrieved from a database. You can achieve this flexibility in the following ways:

- By use of a ServiceLib function that sets locations, as noted in *ServiceLib*; or
- By use of multiple variables, as in the following steps:
  1. Create variables in a service binding library, that are based on the same Service part, and that are bound to different runtime locations.
  2. Declare a client variable that also is based on the Service part.

3. Assign to the client variable one or another of the global variables, depending on conditions at run time. You can place the assignment statement in the **OnException** clause of a **try** block, as that clause will be invoked in response to a failed service invocation; specifically, in response to an exception of type `ServiceInvocationException`.
4. Use the client variable to access the service.

For details on `ServiceInvocationException`, see *EGL system exceptions*.

The use of multiple variables relies on the following rules:

- The variable on the left side of the assignment statement receives a binding in these cases--
  - Only a variable is on the right side. That variable represents a service or interface that was declared in a service binding library or has received a binding in a previous statement.
  - Only a function is on the right side. That function returns a bound interface or service.
- A function parameter receives a binding if the related argument has a binding.

### Build descriptor options

When you generate an EGL service, the build descriptor option **serviceRuntime** is among those used; but the default value is sufficient.

When you generate a Web service, the following build descriptor options are among those used:

- **J2EELevel**
- **serverType**
- **serviceRuntime**
- **WebServiceEncodingStyle**

When you generate a service-binding library that references either a Web service or an interface that is bound to a Web service, the following build descriptor options are among those used:

- **J2EELevel**
- **serverType**
- **serviceRuntime**

### Deployment

Deploying an EGL service is equivalent to deploying a non-J2EE Java application. For details, see *Deploying Java applications outside of J2EE*.

Deploying a Web service is equivalent to deploying a Web application. For details, see *Setting up the J2EE runtime environment for EGL-generated code*.

### Related concepts

“Library part of type `ServiceBindingLibrary`” on page 172

### Related tasks

“Creating an EGL Interface part” on page 150

“Creating an EGL Service part” on page 157

“Creating an Interface part from a Service part” on page 154

“Creating EGL Parts from a WSDL file” on page 154

“Deploying Java applications outside of J2EE” on page 434

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

### Related reference

“Best practices for services and related interfaces in EGL”  
“EGL library ServiceLib” on page 986  
“EGL system exceptions” on page 587  
“Interface part in EGL source format” on page 633  
“Interfaces of type BasicInterface” on page 636  
“J2EELevel” on page 484  
“serverType” on page 487  
“Service part in EGL source format” on page 869  
“serviceRuntime” on page 488  
“webServiceEncodingStyle” on page 498

## Best practices for services and related interfaces in EGL

The following practices are recommended for developing services and related interfaces with EGL:

### Use interfaces as a design tool

You can create one or more interfaces that describe the functionality that you want to have coded in an EGL service. After the interfaces are complete, you or others can code the service, which is said to *implement the interfaces*. The primary meaning is that the service contains every function described in the interfaces. The interfaces provide a kind of contract that the service must fulfill.

This use of interfaces provides the following benefits:

- Helps people in your organization to think clearly about what operations are needed before service development begins
- Allows developers to finish client applications while the service code is under development, as the client code can interact with variables that are based on Interface parts rather than on Service parts

### Avoid unnecessary declarations that refer to a Service or Interface part

Unless you have a reason to do otherwise, use services binding libraries to define *every* variable that refers to a Service or Interface part. You usually do not need to declare those kind of variables elsewhere or to pass them to or return them from a function or to declare more than one for referring to a particular service.

Each of the variables in an EGL library is global to the run unit.

### Create multiple service binding libraries, each for a different collection of service and interface parts

Organize bound variables into small, logical collections to avoid having to regenerate a library as new parts are added.

### Assign a value to the debugImpl field in the @EGLBinding and @WebBinding property (as used in the services binding library)

The property field `debugImpl` identifies a Service part that is accessed at debug time, even if the service is only a development-time stub. If no part is specified, the debug session invokes a deployed service without stepping through that service; but in that case, the absence of a runtime service throws an exception of type `ServiceInvocationException`.

### Use the IN modifier to save time and bandwidth when you do not need to return a value for a given function parameter

Use of the IN modifier reduces the time needed for runtime processing and reduces the amount of data that must be transmitted from the service back to the client.

### **Avoid declaring global variables in a Service part**

If you declare all variables in a service as function parameters or other local variables, the behavior of the service is consistent wherever the service is deployed.

### **Related concepts**

"EGL interfaces" on page 151

"EGL services and Web services" on page 158

### **Related tasks**

"Creating an EGL Interface part" on page 150

"Creating an Interface part from a Service part" on page 154

"Creating an EGL Service part" on page 157

### **Related reference**

"EGL interfaces" on page 151

## **Web transaction support in EGL**

Web transactions, such as VisualAge Generator Web transactions or EGL VGWebTransaction parts, can be used along with EGL pageHandlers in an EGL application. While the recommended application development path is to migrate Web transactions to EGL parts and use pageHandlers for all new parts, it is also possible to create new Web transactions. See *Developing Web transactions in EGL*.

Web transactions can be used with EGL in one or more of these scenarios:

- The recommended method is to migrate VisualAge Generator Web transactions to EGL VGWebTransaction parts and associated EGL parts. These EGL VGWebTransaction parts can then be incorporated into an existing deployment of an EGL application. See *Sources of additional information on EGL* for the migration guide.
- New EGL VGWebTransaction parts are created in an existing EGL Web project. VGWebTransaction parts are a subtype of EGL program parts. See *Program part*.
- VisualAge Generator Web transactions are incorporated into an existing deployment of an EGL application without migrating the VisualAge Generator parts to EGL parts. These Web transactions can forward control to an EGL pageHandler, and the pageHandlers can forward control to a Web transaction. See *Forwarding control between pageHandlers and Web transactions*.

### **Related concepts**

"Sources of additional information on EGL" on page 14

"Developing Web transactions in EGL" on page 165

"Program part" on page 148

"Segmentation in Web transactions" on page 250

"VGWebTransaction part" on page 167

"VGUIRecord part" on page 167

"IDs for Web applications" on page 168

### **Related tasks**

"Configuring a project to run Web transactions" on page 164

"Adding Web transaction support to an EGL Web project" on page 164

"Forwarding control between pageHandlers and Web transactions" on page 165

### **Related reference**

"Gateway servlet parameters" on page 848

"Linkage properties" on page 851

## Configuring a project to run Web transactions

To run VisualAge Generator Web transactions, you must configure the EGL Web project to work correctly with the EGL gateway servlet. Configuring your project in this way requires these steps:

1. Add support for Web transactions to your project. See *Adding Web transaction support to an EGL Web project*.
2. Set the gateway servlet parameters. By default, these parameters are stored in the file `gw.properties` in the folder `JavaResources\JavaSource`. See *Gateway servlet parameters*.
3. Set the linkage properties. These properties are defined in the file specified by the `hptLinkageProperties` parameter in the gateway servlet parameters. By default, the linkage properties are stored in the `csogw.properties` file in the folder `JavaResources\JavaSource`. See *Linkage properties*.

### Related tasks

“Adding Web transaction support to an EGL Web project”

“Forwarding control between pageHandlers and Web transactions” on page 165

### Related reference

“Gateway servlet parameters” on page 848

“Linkage properties” on page 851

## Adding Web transaction support to an EGL Web project

Before you can use Web transactions in your EGL project, you must add support for Web transactions to your project. There are two ways to add support for Web transactions to your Web project:

- Create a new EGL Web project and select the **EGL support with Legacy Web Transaction** check box on the Features page of the New EGL Web Project wizard.
- Add Web transaction support to an existing EGL Web project.

To add support for Web transactions to an existing EGL Web project, follow these steps:

1. In the Project Explorer view, right-click the EGL Web project and then click **Properties**. The Properties window opens.
2. In the Properties window, click **Web Project Features**.
3. Select the **EGL support with Legacy Web Transaction** check box.
4. Click **OK**.

Adding support for Web transactions makes the following changes to the project:

- Jar files that control the EGL gateway servlet are added to the folder `WebContent\WEB_INF\lib`. These files are also added to the project's Java build path.
- JSP and GIF files are added to the project's WebContent folder.
- Files that set properties for the gateway servlet are added to the folder `JavaResources\JavaSource`.
- The gateway servlet is registered in the Web configuration file.

You can not remove support for Web transactions from a project.

### Related tasks

“Configuring a project to run Web transactions”



### Related reference

“Gateway servlet parameters” on page 848

“Linkage properties” on page 851

## Developing Web transactions in EGL

Web transactions work in a fundamentally different way than pageHandlers do. Like a PageHandler, a Web transaction interacts with a Web page. Unlike a PageHandler, a Web transaction depends on a VGUIRecord part to make that interaction possible.

The VGUIRecord part is the model for the data to be shown on the Web page and the data to be collected from the user, if any. For this reason, creating a Web transaction in EGL requires you to create and coordinate two separate but related EGL parts: an EGL program part of type VGWebTransaction and an EGL VGUIRecord part.

Web transaction development in EGL usually involves the following steps:

1. Create an EGL program of type VGWebTransaction.
2. Create an EGL VGUIRecord part.
3. In the VGUIRecord part, define the data structures and control items to be shown on the page.
4. In the VGWebTransaction, create a variable from the VGUIRecord part.
5. In a function within the VGWebTransaction, reference the VGUIRecord variable with the **converse** or **show** statement. These statements present the Web page, using the data in the VGUIRecord.

Once you have created these parts, you must configure them to work together, along with the JSP that is generated along with the VGUIRecord part.

### Related concepts

“Segmentation in Web transactions” on page 250

“VGUIRecord part” on page 167

“VGWebTransaction part” on page 167

“IDs for Web applications” on page 168

### Related tasks

“Configuring a project to run Web transactions” on page 164

### Related reference

“VGWebTransaction program in EGL source format” on page 847

“VGUIRecord part in EGL source format” on page 1089

**Forwarding control between pageHandlers and Web transactions:** EGL pageHandlers, VisualAge Generator Web transactions, and EGL VGWebTransaction parts can forward control and data to each other, but each forwards control and data in a different way.

To forward control from a pageHandler to a VisualAge Generator Web transaction or EGL VGWebTransaction part, use the EGL **forward** statement.

- The simplest way to forward control to a Web transaction or VGWebTransaction part is to forward control to the gateway servlet instead of the Web transaction or VGWebTransaction part directly. In this case, the **forward** statement sends the browser to the gateway servlet’s introductory page, and it is not possible to send any parameters. This type of **forward** statement has the following format:



forward to URL "URL";

*URL*

The complete URL of the gateway servlet on which the target Web transaction is located.

- To forward control directly to a specific VisualAge Generator Web transaction or EGL VGWebTransaction part, you must specify which Web transaction or VGWebTransaction that the gateway servlet should load. This type of **forward** statement has the following format:

forward to URL "URL?hptAppId=linkageID";

*URL*

The complete URL of the gateway servlet on which the target Web transaction is located.

*linkageID*

The linkage ID of the Web transaction or VGWebTransaction part. This parameter is the *servername* parameter in the application property that defines this Web transaction in the linkage properties file. For example, if a Web transaction is defined in the linkage properties file with the code application.WEBUITRAN=CICS5, the *linkageID* parameter in the **forward** statement is CICS5. See *Linkage properties*.

- To forward control and parameters to a VisualAge Generator Web transaction or EGL VGWebTransaction part, you must append pairs of parameters and values to the end of the URL. This type of **forward** statement has the following format:

forward to URL "URL?hptAppId=linkageID  
&parameter1=value1";

*URL*

The complete URL of the gateway servlet on which the target Web transaction is located.

*linkageID*

The linkage ID of the Web transaction or VGWebTransaction part. This linkage ID is the *servername* parameter in the application property that defines this Web transaction in the linkage properties file. For example, if a Web transaction is defined in the linkage properties file with the code application.WEBUITRAN=CICS5, the *linkageID* parameter in the **forward** statement is CICS5. See *Linkage properties*.

*parameter1*

The name of the parameter to pass.

*value1*

The value of the parameter.

Additional parameter-value pairs can be passed if they are separated by the and symbol (&). For example, this **forward** statement passes two parameters in addition to the parameter that specifies the target Web transaction:

forward to URL "http://localhost:9080/myProjectName/myGatewayServletName  
&hptAppId=myWebTransaction&myParam1=myvalue1&myParam2=myvalue2";

To forward control from a VisualAge Generator Web transaction or EGL VGWebTransaction part to an EGL pageHandler part, you must modify the JSP file associated with the Web transaction.

- If the JSP file associated with the Web transaction uses a **form** tag, modify the **action** attribute of that tag to reference the URL of the JSP file associated with the pageHandler.

- If the JSP file associated with the Web transaction uses a **anchor** tag, modify the **href** attribute of that tag to reference the URL of the JSP file associated with the pageHandler.
- To pass parameters to a pageHandler, use the same method as in passing parameters from a pageHandler to a Web transaction. If the pageHandler is set to receive parameters in its onPageLoad function, you can add those parameters to the end of the URL in name-value pairs separated by the and symbol (&).

#### Related tasks

“Configuring a project to run Web transactions” on page 164

“Adding Web transaction support to an EGL Web project” on page 164

#### Related reference

“forward” on page 686

“Linkage properties” on page 851

“PageHandler part in EGL source format” on page 785

**VGWebTransaction part:** A VGWebTransaction part is a generatable logic part. Like a pageHandler, a VGWebTransaction controls the user’s runtime interaction with a Web page. Unlike a pageHandler, a VGWebTransaction relies on one or more VGUIRecord parts to communicate with the Web page and show HTML fields and control structures. The fields in these VGUIRecords define the input and output fields and the links and buttons that will be shown on the Web page

The VGWebTransaction part uses VGUIRecords in these ways:

- To receive data from a user’s Web-page submission, the VGWebTransaction must reference a VGUIRecord part in the VGWebTransaction’s **inputUIRecord** property. See *Program part properties*.
- The VGWebTransaction must create one or more variables of type VGUIRecord and reference these variables in each **converse** or **show** statement that presents the Web page. The Web page is rendered according to the fields in the VGUIRecord parts.

#### Related concepts

“Developing Web transactions in EGL” on page 165

“VGUIRecord part”

“IDs for Web applications” on page 168

#### Related tasks

“Forwarding control between pageHandlers and Web transactions” on page 165

#### Related reference

“VGWebTransaction program in EGL source format” on page 847

“Program part properties” on page 856

**VGUIRecord part:** A VGUIRecord part is a generatable part and is the basis of a VGUI record, which is a VGWebTransaction program or function variable that makes communication possible between the program and a specific Web page. A VGUIRecord is referenced in each **converse** or **show** statement that presents the Web page.

The VGUIRecord part is generated into the following outputs, which are deployed in the Web application server:

- The *VGUI record object* is a Java class instance that contains the data being passed between the program and the Web page.

- The *VGUI record bean* is a Java bean that provides data validation and event-handling services. The UI record bean is used in these ways--
  - When a JSP prepares a Web page for display, the JSP accesses the VGUI record bean, which accesses the data in the VGUI record object.
  - When user input is returned from a Web browser, the bean validates the input either for use by business logic or (if validation failed) for subsequent re-display. After a successful validation, the bean stores the data in the VGUI record bean, and the EGL runtime makes the data available to the program.

Characteristics of the VGUI record fields (length, primitive type) must match characteristics of Web page fields. Fields in the VGUIRecord part can have the property **uiType**, which specifies the type of HTML tag to create for that field. Depending on the **uiType** property, a field in the VGUIRecord part can appear as an input field, an output field, or as a control button.

When you design the JSP that presents the Web page, you need to ensure that fields in the Web page are *bound* to the equivalent structure fields in the VGUIRecord part. This binding means that the JSF run time will transfer data between a given field in the page and the equivalent field in the VGUI record bean.

#### Related concepts

“Developing Web transactions in EGL” on page 165

“VGWebTransaction part” on page 167

#### Related reference

“VGUIRecord part in EGL source format” on page 1089

“uiType” on page 829

**IDs for Web applications:** In relation to users of VGWebTransaction programs, EGL supplies two unique IDs:

- A *session ID*, which identifies the Web application server session. This ID is stored in the system variable **sysVar.sessionID**.
- One or more *conversation IDs*, each specific to an execution thread that runs a sequence of program proxies in a given Web application server session. The conversation ID is unchanged when a program in a Web application is invoked by way of a **converse** statement or by way of a **show** statement that has a returning clause. A new conversation ID is assigned, however, when the user invokes a program in response to a forward statement that has no returning clause.

The conversation ID is stored in **sysVar.conversationID**.

You can use either or both of the IDs as a key for storing application data.

#### Related concepts

“Web transaction support in EGL” on page 163

“Developing Web transactions in EGL” on page 165

“VGWebTransaction part” on page 167

“conversationID” on page 1067

“sessionID” on page 1071

---

## Creating an EGL library part

An EGL library part contains a set of functions, variables, and constants that can be used by programs, PageHandlers, or other libraries. To create an EGL library part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Library**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the library name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example `myLibrary`.
4. Select the type of library by clicking one of the following radio buttons:
  - **Basic - Create a basic library**
  - **Native - Create a native library**
5. Click the **Finish** button.

### Related concepts

“EGL projects, packages, and files” on page 15

“Introduction to EGL” on page 1

“Library part of type `basicLibrary`”

“Library part of type `basicLibrary`”

### Related tasks

“Creating an EGL source folder” on page 129

“Creating an EGL Web project” on page 127

### Related reference

“Creating an EGL source file” on page 130

“Library part in EGL source format” on page 756

“Naming conventions” on page 778

## Library part of type `basicLibrary`

A *library part* of type `basicLibrary` contains a set of functions, variables, and constants that can be used by programs, PageHandlers, or other libraries. It is recommended that you use libraries to maximize your reuse of common code and values.

The type specification *basicLibrary* indicates that the part is generated into a compilable unit and includes EGL values and code for local execution. This type is the default when the keyword **type** is not specified. For details on creating a library to access a native DLL from an EGL-generated Java program, see *Library part of type `nativeLibrary`*.

Rules for a library of type *basicLibrary* are as follows:

- You can reference a library’s functions, variables, and constants without specifying the library name, but only if you include the library in a program-specific Use declaration.
- Library functions can access any system variables that are associated with the invoking program or PageHandler. The following rules apply:

- When a function in a library receives a record as an argument, the record cannot be used for input or output (I/O) or for testing an I/O state such as `endOfFile`. The code that invokes the library, however, can use the record in either way.
- When you declare a record in a library, the library-based functions can use the record for input or output (I/O) and for testing the I/O state (for end of file, for example). The code that invokes the library, however, cannot use the record in either way.
- Library functions can use any statements except these:
  - `converse`
  - `forward`
  - `show`
  - `transfer`
- A library cannot access a text form.
- A library that accesses a print form must include a use declaration for the related form group.
- You can use the modifier **private** on a function, variable, or constant declaration to keep the element from being used outside the library.
- Library functions that are declared as public (as is the default) are available outside the library and cannot have parameters that are of a loose type, which is a special kind of primitive type that is available only if you wish the parameter to accept a range of argument lengths. For details on the loose type, see *Function part in EGL source format*.
- For COBOL output, the length of the generated name for the library and for any public library functions must be eight characters or less.

The library is generated separately from the parts that use it. EGL run time accesses the library part by using the setting of the library property **alias**, which defaults to the EGL library name.

At run time, the library is loaded when first used and is unloaded when the program or PageHandler that accessed the library leaves memory, as occurs when the run unit ends..

A PageHandler gets a new copy of the library whenever the PageHandler is loaded. Also, a library that is invoked by another library remains in memory as long as the invoking library does.

A library that is used only for its constants is not loaded at run time because constants are generated as literals in the programs and PageHandlers that reference them.

#### **Related concepts**

“Library part of type `basicLibrary`” on page 169

“Library part of type `ServiceBindingLibrary`” on page 172

“Run unit” on page 866

“Segmentation in text applications” on page 189

“`show`” on page 751

“`transfer`” on page 752

“Use declaration” on page 1091

#### **Related reference**

“`converse`” on page 672

“forward” on page 686  
 “Function part in EGL source format” on page 621  
 “Library part in EGL source format” on page 756  
 “Run unit” on page 866  
 “Segmentation in text applications” on page 189  
 “show” on page 751  
 “transfer” on page 752  
 “Use declaration” on page 1091

## Library part of type nativeLibrary

A library of type `nativeLibrary` enables your EGL-generated Java code to invoke a single, locally running DLL. The code for that DLL is not written in the EGL language. For information on developing a basic library, which contains shared functions and values that *are* written in the EGL language, see *Library part of type basicLibrary*.

In a library of type `nativeLibrary`, the purpose of each function is to provide an interface to a DLL function. You cannot define statements in the EGL function, and you cannot declare variables or constants anywhere in the library.

The EGL runtime accesses a DLL-based function by using the setting of the EGL function property **alias**, which defaults to the EGL function name. Set that property explicitly if the name of the DLL-based function does not conform to the conventions described in *Naming conventions*.

The library property **callingConvention** specifies how the EGL runtime passes data between the two kinds of code:

- The EGL code that invokes the library function; and
- The function in the DLL.

The only value now available for **callingConvention** is *I4GL*:

- Data is passed in accordance with the Informix stack format. Each input parameter is placed on an input stack, and each output parameter is placed on an output stack.
- You cannot pass arguments such as records or dictionaries. Only these are valid:
  - Primitive variables, including variables of type ANY
  - Fields that are in dataTables, print forms, text forms, and fixed records, but only if the field lacks a substructure

The parameters in the library functions must be primitive variables and may be of type ANY, but cannot be of a loose type and cannot include the **field** modifier.

The library property **dllName** specifies the DLL name, which is final; it cannot be overridden at deployment time. If you do not specify a value for the library property **dllName**, you must specify the DLL name in the Java runtime property `vgj.defaultI4GLNativeLibrary`. Only one such Java runtime property is available for a run unit, so only one DLL can be specified, aside from DLLs that are identified in the EGL libraries.

Whether you specify the DLL name at development time (in **dllName**) or at deployment time (in `vgj.defaultI4GLNativeLibrary`), the DLL must reside in the directory path identified in a runtime variable; that variable is either `PATH` (on Windows 2000/NT/XP) or `LIBPATH` (on UNIX platforms).

Library functions are automatically declared as public to ensure that they are available outside the library. In your other EGL code, you can reference a function by its function-alias name alone, without specifying the library name, but only if you include the library in a program-specific Use declaration.

The EGL library is generated as a Java class that is separate from the code that accesses the library and from the DLL. The EGL runtime accesses that class by using the setting of the library property **alias**, which defaults to the EGL library name. Set that property explicitly if the name of the library part does not conform to Java conventions.

At run time, a DLL is loaded when first used and is unloaded when the accessing program or PageHandler leaves memory, as occurs when the run unit ends.

A PageHandler gets a new copy of the DLL whenever the PageHandler is loaded. Also, a DLL that is invoked by an EGL library of type basicLibrary remains in memory as long as the invoking library does.

The following native library provides access to a DLL written in C:

```
Library myLibrary type nativeLibrary
{callingConvention="I4GL", dllname="mydll"}

Function entryPoint1( p1 int nullable in,
                     p2 date in, p3 time in,
                     p4 interval in, p5 any out)
end

Function entryPoint2( p1 float in,
                     p2 String in,
                     p3 smallint out)
end

Function entryPoint3( p1 any in,
                     p2 any in,
                     p3 any out,
                     p4 CLOB inout)
end
end
```

#### **Related concepts**

"Java runtime properties" on page 431

"Library part of type basicLibrary" on page 169

#### **Related reference**

"Function part in EGL source format" on page 621

"Java runtime properties (details)" on page 642

"Library part in EGL source format" on page 756

"Naming conventions" on page 778

"Run unit" on page 866

"Use declaration" on page 1091

## **Library part of type ServiceBindingLibrary**

A service binding library (a library part of type ServiceBindingLibrary) contains only a set of variables, with no functions or constants:

- Each variable is of type Interface (subtype BasicInterface) or Service
- Each declaration has binding information, which includes the details needed to relate the variable with a particular service implementation at run time



You can declare variables of type `Interface` or `Service` elsewhere in EGL; but the declaration can include binding information only in a service binding library. Each declaration in this type of library includes one of the following complex properties:

#### **@EGLBinding**

For binding a variable to an EGL service. Details are in “@EGLBinding.”

#### **@WebBinding**

For binding a variable to a Web service. Details are in “@WebBinding” on page 174.

The library of type `serviceBinding` includes the property **runtimeBind**, which is set to *no* (the default) or *yes*. If you set the property to *yes*, a library-specific property file is created at generation time. That file includes binding information that takes effect at run time and can be changed at deployment time. Details are in “Service binding library property file” on page 174.

Any EGL library is generated separately from the parts that use it. EGL run time accesses the library by using the setting of the library property **alias**, which defaults to the EGL library name. That property is described in *Library part in EGL source format*.

At run time, the library is loaded when first used and is unloaded when the code that accessed the library leaves memory, as occurs when the run unit ends..

A `PageHandler` gets a new copy of the library whenever the `PageHandler` is loaded. Also, a library that is invoked by another library remains in memory as long as the invoking library does.

For details on creating a library to hold EGL values and code for local execution, see *Library part of type BasicLibrary*. For details on creating a library to access a native DLL from an EGL-generated Java program, see *Library part of type NativeLibrary*.

#### **@EGLBinding**

The complex property `@EGLBinding` specifies the binding for a variable that refers to an EGL service. The property fields and their types are as follows:

##### **commType CommTypeKind**

The type of communications used to connect the client and service. One of the following values:

##### **LOCAL**

As appropriate if the client and service are in the same run unit

##### **TCPIP**

As appropriate if the client and service communicate by way of TCP/IP

The property field **commType** is required.

##### **serviceName STRING**

The name of the Service part to which the variable refers.

The property field **serviceName** is required.

##### **servicePackage STRING**

The name of the EGL package in which the Service part resides.

The property field **servicePackage** is required.



**serviceAlias STRING**

The alias (if any) of the EGL Service part.

**tcpipLocation STRING**

A string having the following format:

`host:portNumber`

*host* is the TCP/IP host name that refers to the machine where the service runs.  
*portNumber* is the number of the TCP/IP port that provides access to the service.

The property field **tcpipLocation** is required if the value of **commType** is *TCPIP*.

**debugImpl STRING**

The name of a Service part that is accessed at debug time. If no part is specified, the debug session invokes the deployed service but does not step through that service.

**@WebBinding**

The complex property @WebBinding specifies the binding for a Web service. The property fields are as follows:

**wsdlFile STRING**

The location of the WSDL file on the client side, with the start of the path in the WebContent folder, as in the following example:

`WebContent/WEB-INF/wsdl>HelloWorld.wsdl`

The property field **wsdlFile** is required.

**wsdlService STRING**

The WSDL service element name.

This property field is required.

**wsdlPort STRING**

The WSDL port element name.

This property field is required.

**endPoint STRING**

The URL of the Web service. The default is the value of the soap:address element, as specified in the WSDL file.

**debugImpl STRING**

The name of a Service part that is accessed at debug time. If no part is specified, the debug session invokes the deployed service but does not step through that service.

**Service binding library property file**

The property file is composed a series of file property and value pairs. Only the following file properties are supported:

```
egl.service.varName.eglBinding.commType
egl.service.varName.eglBinding.servicePackage
egl.service.varName.eglBinding.serviceAlias
egl.service.varName.eglBinding.tcpipLocation
```

```
egl.service.varName.webBinding.endpoint
```

As shown, the name of each of file property begins with `egl.service` and continues with the following format:

*.varName.bindingType.propertyName=value*

*varName*

Name of the variable of type Service or Interface

*bindingType*

One of the following:

- **eglBinding** (if the variable refers to an EGL service)
- **webBinding** (if the variable refers to a Web service)

*propertyName*

A binding property such as **commType** or **endpoint**. Not all binding properties are supported in the properties file, as noted earlier.

*value*

The value of the binding property.

#### **Related concepts**

"EGL interfaces" on page 151

"EGL services and Web services" on page 158

#### **Related tasks**

"Creating EGL Parts from a WSDL file" on page 154

#### **Related reference**

"Best practices for services and related interfaces in EGL" on page 162

"Library part in EGL source format" on page 756

---

## **Creating an EGL dataTable part**

An EGL dataTable part associates a data structure with an array of initial values for the structure. To create an EGL dataTable part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Data Table**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the dataTable name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myDataTable. Select a dataTable sub-type (for details, see *Data Table part in EGL source format*).
4. Click the **Finish** button.

#### **Related concepts**

"DataTable" on page 176

"EGL projects, packages, and files" on page 15

"Introduction to EGL" on page 1

#### **Related tasks**

"Creating an EGL source folder" on page 129

"Creating an EGL Web project" on page 127

#### **Related reference**

"Creating an EGL source file" on page 130

"DataTable part in EGL source format" on page 568

"Naming conventions" on page 778

## DataTable

An EGL *DataTable* is primarily composed of these components:

- A structure, with each top-level item defining a column.
- An array of values that are consistent with those columns. Each element of that array defines a row.

A *DataTable* of error messages, for example, might include these components:

- The declaration of a numeric field and a character field
- A list of paired values like these—

```
001   Error 1
002   Error 2
003   Error 3
```

You do not declare a *DataTable* as if you were declaring a record or data item. Instead, any code that can access a *DataTable* can treat that part as a variable. For details on part access, see *References to parts*.

Any code that can access a *DataTable* has the option of referencing the part name in a Use declaration.

### Types of DataTables

Some types of *DataTables* are for runtime validation; specifically, to hold data for comparison against form input. (You relate the *DataTable* to the input field when you declare the form part.) Three types of validation *DataTables* are available:

#### **matchValidTable**

The user's input must match a value in the first *DataTable* column.

#### **matchInvalidTable**

The user's input must be different from any value in the first *DataTable* column.

#### **rangeChkTable**

The user's input must match a value that is between the values in the first and second column of at least one *DataTable* row. (The range is inclusive; the user's input is valid if it matches a value in the first or second column of any row.)

The other types of *DataTables* are as follows:

#### **msgTable**

Contains runtime messages.

#### **basicTable**

Contains other information that is used in the program logic; for example, a list of countries and related codes.

### DataTable generation

The output of *DataTable* generation is a pair of files, each named for the *DataTable*. One file has the extension *.java*, the other has the extension *.tab*. The *.tab* file is not processed by the Java compiler, but is included in the root of the directory structure that contains the package. If the package is *my.product.package*, for example, the directory structure is *my/product/package*, and the *.tab* file is in the directory that contains the subdirectory *my*.

You do not need to generate *DataTables* if you are generating into a package to which you had previously generated the same *DataTables*.

To save generation time when you do not need to generate DataTables, assign NO to the build descriptor option **genTables**.

### Properties of the DataTable

You can set the following properties:

- An **alias** is incorporated into the names of generated output. If you do not specify an alias, the part name is used instead.
- The **shared** property indicates whether multiple users can access the DataTable. The default is *no*.
- The **resident** property indicates whether the DataTable remains in memory even when no program is using the DataTable. (The program goes into memory when first accessed.) The default is *no*. You can specify *yes* only if the shared specification is also *yes*.

### Related concepts

“References to parts” on page 23

### Related reference

“DataTable part in EGL source format” on page 568

“Use declaration” on page 1091



---

## Inserting code snippets into EGL and JSP files

The Snippets view lets you insert reusable programming objects into your code. The Snippets view contains several pieces of EGL code, as well as code for many other technologies. You can use the snippets provided or add your own to the Snippets view. For more information about using the Snippets view, see *Snippets view*.

To insert an EGL code snippet into your code, do as follows:

1. Open the file to which you want to add a snippet.
  2. Open the Snippets view.
    - a. Click **Window > Show View > Other**.
    - b. Expand **Basic** and click **Snippets**.
    - c. Click **OK**.
  3. In the Snippets view, expand the **EGL** drawer. This drawer contains the available EGL code snippets.
  4. Use one of these methods to insert a snippet into the file:
    - Click and drag a snippet into the source code.
    - Double-click a snippet to insert that snippet at the current cursor position. You may see a window describing the variables in the snippet. If so, enter values for these variables and then click **Insert**.
- Note:** If the cursor turns into a circle with a strike through it, indicating that the snippet can not be inserted at that point, you may be trying to insert the snippet into the wrong place. Check the snippet's details to find out where it should be inserted in the code.
5. Change the pre-defined names of functions, variables, and data parts in the snippet as appropriate to your code. Most snippets include comments that explain what names need to be changed.

Following are the snippets available in EGL:

*Table 6. Snippets available in EGL*

Snippet name	Description
setCursorFocus	A JavaScript function that sets the cursor focus to a specified form field on a Web page.
autoRedirect	A JavaScript function that tests for the presence of a session variable. If the session variable is not present, it forwards the browser to a different page.
getClickedRowValue	An EGL function that retrieves the hyperlinked value of a clicked row in a data table.
databaseUpdate	An EGL function that updates a single row of a relational table when passed a record from a PageHandler.

## Related concepts

Snippets view

## Related tasks

“Using the EGL templates with content assist” on page 131

“Setting the focus to a form field”

“Testing browsers for a session variable”

“Retrieving the value of a clicked row in a data table” on page 181

“Updating a row in a relational table” on page 182

---

## Setting the focus to a form field

The `setCursorFocus` snippet in the JSP drawer of the Snippets view is a JavaScript function that sets the cursor focus to a specified form field on a Web page. It must be placed within a `<script>` tag in a JSP page. To insert and configure this snippet, follow these directions:

1. Insert the snippet's code into the source code of the page. For more information, see *Inserting EGL code snippets*.
2. Replace `[n]` with the number of the form field which will receive focus. The form fields are numbered beginning with zero. For example, use `[3]` to set focus to the fourth field on the page.
3. Set the form name to `form1`.
4. Change the `<body>` tag of the JSP page to `<body onload="setfocus();">`.

The code inserted by this snippet is as follows:

```
function setFocus() {  
    document.getElementById('form1').elements[n].select();  
    document.getElementById('form1').elements[n].focus();  
}
```

## Related tasks

“Inserting code snippets into EGL and JSP files” on page 179

---

## Testing browsers for a session variable

The `autoRedirect` snippet in the JSP drawer of the Snippets view tests for the presence of a session variable. If the session variable is not present, the customized code forwards control to a different Web page.

The snippet must be placed within the `<head>` tag of a JSP page after the `<pageEncoding>` tag. To insert and configure this snippet, follow these directions:

1. From the JSP drawer of the Snippets view, add the snippet to the `<head>` tag of the page after the `<pageEncoding>` tag. For more information, see *Inserting code snippets into EGL and JSP files*. The Insert Template window opens.
2. In the Insert Template window, set `SessionAttribute` to the name of the session variable that is being tested. The default value is `UserID`.
3. Set `ApplicationName` to the name of your project or application. The default value is `EGLWeb`.
4. Set `PageName` to the name of the page that the browser will be redirected to if the session variable is absent. The default value is `Login.jsp`.
5. When you have customized the values in the Insert Template window, click **Insert**.

The code inserted by this snippet is as follows:

```
<%
if ((session.getAttribute("userID") == null ))
{
    String redirectURL =
    "http://localhost:9080/EGLWeb/faces/Login.jsp";
    response.sendRedirect(redirectURL);
}
%>
```

#### Related tasks

“Inserting code snippets into EGL and JSP files” on page 179

---

## Retrieving the value of a clicked row in a data table

The `getClickedRowValue` snippet in the EGL drawer of the Snippets view is a function that retrieves the hyperlinked value of a clicked row in a data table. This snippet must be placed in an EGL PageHandler. This snippet has the following prerequisites:

1. The JSP page has a data table.
2. The names of the JSP identifiers have not been changed from the default.
3. The page is defined as request in scope in `faces-config.xml`, not session.

To insert and configure this snippet, follow these directions:

1. Insert the snippet’s code into the PageHandler. For more information, see *Inserting EGL code snippets*.
2. Define a char or string variable to receive the clicked value.
3. Add a command hyperlink (from the Faces Components drawer in the Palette view) to a field in the data table.
4. For the target of the command hyperlink, specify the name of the JSP page. The hyperlink links to its own page.
5. Add a parameter to the hyperlink and give that parameter the same name as the variable in the PageHandler that receives the clicked value.
6. Set the action property (located on the All tab of the Properties view) to the `getVal()` function.

The code inserted by this snippet is as follows:

```
function getVal()
    javaLib.store((objId)"context",
        "javax.faces.context.FacesContext",
        "getCurrentInstance");
    javaLib.store((objId)"root",
        (objId)"context", "getViewRoot");
    javaLib.store((objId)"parm",
        (objId)"root",
        "findComponent",
        "form1:table1:param1");
    recVar = javaLib.invoke((objId)"parm",
        "getValue");
end
```

#### Related tasks

“Inserting code snippets into EGL and JSP files” on page 179



---

## Updating a row in a relational table

The databaseUpdate snippet in the EGL drawer of the Snippets view is a function that updates a single row of a relational table when passed a record from a PageHandler. This snippet is intended to be placed in an EGL library. To insert and configure this snippet, follow these directions:

1. Insert the snippet's code into the PageHandler. For more information, see *Inserting EGL code snippets*.
2. Replace {tableName} and {keyColumn} with the name of the table and its primary key column.

The code inserted by this snippet is as follows:

```
Function updateRec(${TableName}New ${TableName})

    // Function name - call this function
    // passing the ${TableName} Record as a parameter
    ${TableName}Old ${TableName};

    // A copy of the Record, used
    // to lock the table row and to obtain
    // the existing row values prior to update try
    ${TableName}Old.${KeyColumn} =
        ${TableName}New.${KeyColumn};
    get ${TableName}Old forUpdate;

    // Get the existing row.
    // Note that if you had custom processing to do,
    // you would insert your code after this call
    move ${TableName}New to ${TableName}Old byName;

    //Move the updated values to the copy-row
    replace ${TableName}Old;

    //And replace the row in the database.
    sysLib.commit();

    //Commit your changes to the Database
    onException
        //If the update fails...
        sysLib.rollback();

        // cancel all database updates
        // (assuming this is permitted
        // by your database) and call
        // a custom error handling routine
    end
end
```

### Related tasks

"Inserting code snippets into EGL and JSP files" on page 179

---

## Working with text and print forms

---

### Creating an EGL formGroup part

An EGL formGroup part defines a collection of text and print forms. To create an EGL formGroup part, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Form Group**.
3. Select the project or folder that will contain the EGL file, then select a package. Since the formGroup name will be identical to the file name, choose a file name that adheres to EGL part name conventions. In the EGL Source File Name field, type the name of the EGL file, for example myFormGroup.
4. Click the **Finish** button.

#### Related concepts

"EGL projects, packages, and files" on page 15

"EGL form editor overview" on page 194

"Editing form groups with the EGL form editor" on page 195

"FormGroup part"

"Introduction to EGL" on page 1

#### Related tasks

"Creating an EGL source folder" on page 129

"Creating an EGL Web project" on page 127

#### Related reference

"Creating an EGL source file" on page 130

"Form part in EGL source format" on page 606

"Naming conventions" on page 778

### FormGroup part

An EGL FormGroup part serves two purposes:

- Defines a collection of text and print forms. (Forms that are unique to the part are defined within the part or are included by way of a Use declaration. Forms that are common to several FormGroup parts are included by way of a Use declaration.)
- Defines zero to many *floating areas*, as described in *Form part*

You do not declare a FormGroup part as if you were declaring a record or dataItem. Instead, your program accesses a FormGroup part (and the related forms) only if the following statements apply:

- The location of the FormGroup part is accessible to the program, as described in *References to parts*
- A Use declaration in the program references the FormGroup part

A program can include no more than two FormGroup parts; and if two are specified, one must be a *help group*. A help group contains one or more *help forms*, which are read-only forms that give information in response to a user keystroke.

Forms are available at run time only if you generate the FormGroup part. The generated output for Java is a class for the FormGroup part and a class for each Form part. The generated output for a COBOL program is as follows:

- Text forms are generated into an object module
- Print forms are generated into a printing-services program

At preparation time, each of those entities is processed into a separate runtime load module. The EGL runtime handles the interaction of your generated program and the form-specific code.

Form parts cannot be generated separately.

#### **Related concepts**

"Form part"

"References to parts" on page 23

"Editing form groups with the EGL form editor" on page 195

#### **Related tasks**

"Creating an EGL formGroup part" on page 183

#### **Related reference**

"Use declaration" on page 1091

## **Form part**

A *form part* is a unit of presentation. It describes the layout and characteristics of a set of fields that are shown to the user at one time.

You do not declare a form as if you were declaring a record or dataItem. To access a form part, your program must include a use declaration that refers to the related form group.

A form part is of one of two types, *text* or *print*:

- A form of type *text* defines a layout that is displayed on a 3270 screen or in a command window. With one exception, any text form can have both constant fields and variable fields, including variable fields that accept user input. The exception is a *help form*, which is solely for presenting constant information.
- A form of type *print* defines a layout that is sent to a printer. Any print form can have both constant and variable fields.

Form properties determine the size and position of the output on a screen or page and specify formatting characteristics of that output.

A given form can be displayed on one or more *devices*, each of which is an output peripheral or is the operational equivalent of an output peripheral:

- A *screen device* is a terminal, monitor, or terminal emulator. The output surface is a screen.
- A *print device* is a file that can be sent to a printer or is the printer itself. The output surface is a page.

Whether of type *text* or *print*, a form is further categorized as follows:

- A *fixed form* has a specific starting row and column in relation to the output surface of the device. You could assign a fixed print form, for example, to start at line 10, column 1 on a page.

- A *floating form* has no specific starting row or column; instead, the placement of a floating form is at the next unoccupied line in an output surface sub-area that you declare. The declared sub-area is called a *floating area*.

You might declare a floating area to be a rectangle that starts at line 10, extends through line 20, and is the maximum width of the output device. If you have a one-line floating form of the same width, you can construct a loop that acts as follows for each of 20 times:

1. Places data in the floating map
2. Writes the floating map to the next line in the floating area

One or more floating areas are declared in the FormGroup part, but only one can accept floating forms for a particular device. If you try to present a floating form in the absence of a floating area, the entire output surface is treated as a floating area.

- A *partial form* is smaller than the standard size of the output surface for a particular device. You can declare and position partial forms so that multiple forms are displayed at different horizontal positions. Although you can specify the starting and ending columns for a partial form, you cannot display forms that are next to one another.

Additional details are specific to the form type:

- Print forms
- Text forms

#### **Related concepts**

"Print forms" on page 186

"Text forms" on page 188

"Editing form groups with the EGL form editor" on page 195

"Form templates in the EGL form editor" on page 200

#### **Related tasks**

"Creating a form in the EGL form editor" on page 196

#### **Related reference**

"FormGroup part in EGL source format" on page 603

"Form part in EGL source format" on page 606

## **Creating an EGL print form**

A print form is an EGL form part that defines a layout to send to a printer. To create an EGL print form, do as follows:

1. Identify an EGL file to contain the print form and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the print form according to EGL syntax (for details, see *Form part in EGL source format*). You can use content assist to place an outline of the form part syntax in the file.
3. Save the EGL file.

#### **Related concepts**

"EGL projects, packages, and files" on page 15

"Form part" on page 184

"Print forms" on page 186

"Editing form groups with the EGL form editor" on page 195

### Related tasks

"Creating an EGL source file" on page 130

"Using the EGL templates with content assist" on page 131

"Creating a form in the EGL form editor" on page 196

### Related reference

"Content assist in EGL" on page 577

"Form part in EGL source format" on page 606

"Naming conventions" on page 778

## Print forms

Forms and their types are introduced in *Form part*. The current page outlines how to present print forms.

**Print process:** Printing is a two-step process:

- First, you code **print** statements, each of which adds a form to a runtime buffer
- Next, the EGL runtime adds the symbols needed to start a new page, sends all the buffered forms to a print device, and erases the contents of the buffer. Those services are provided in response to any of the following circumstances:
  - The program runs a **close** statement on a print form that is destined for the same print device; or
  - The program is in segmented mode (as described in *Segmentation*) and runs a **converse** statement; or
  - The program was called by a non-EGL (and non-VisualAge Generator) program, and the called program ends; or
  - The main program in the run unit ends.

In the case of multiform output, the **print** statements must be invoked in the order in which you want to present the forms. Consider the following example:

- At the top of the output, a fixed form identifies a purchasing company and an order number
- In a subsequent floating area, a series of identically formatted floating forms identify each item of the company's order
- At the bottom of the output, a fixed form indicates the number of screens or pages needed to scroll through the list of items

You can achieve that output by submitting a series of **print** statements that each operate on a print form. Those statements reference the forms *in the following order*:

1. Top form
2. Floating form, as presented by a **print** statement that is invoked repeatedly in a loop
3. Bottom form

The symbols needed to start a new page are inserted in various circumstances, but you can cause the insertion by invoking the system function `ConverseLib.pageEject` before issuing a **print** statement.

**Considerations for fixed forms:** The following statements apply to fixed forms:

- If you issue a print statement for a fixed form that has a starting line greater than the current line, EGL inserts the symbols needed to advance the print device to the specified line. Similarly, if you issue a print statement for a fixed form that has a starting line less than the current line, EGL inserts the symbols needed to start a new page.

- If a fixed form overlays *some but not all* lines in another fixed form, EGL automatically inserts the symbols needed to start a new page and places the second fixed form on the new page.
- If a fixed form overlays *all* lines in another fixed form, EGL replaces the existing form without clearing the rest of the output from the buffer. To keep the existing output and place the new form on the next page, invoke the system function `ConverseLib.pageEject` before issuing the **print** statement for the new form.

**Considerations for floating forms:** The following mistakes can occur if you are using floating forms:

- You issue a **print** statement to place a floating form beyond the end of the floating area; or
- You issue a **print** statement that at least partially overlays a floating area with a fixed form, then issue a **print** statement to add a floating form to the floating area.

The result in either case is that EGL inserts the symbols needed to start a new page, and the floating form is placed on the first line of the floating area on the new page. If the page is similar to the order-and-item output described earlier, for example, the new page does not include the topmost fixed form.

**Print destination:** When EGL processes a **close** statement to present a print file, the output is sent to a printer or data set. You can specify the destination at any of three times:

- At test time (as described in *EGL debugger*)
- At generation time (as described in *Resource associations and file types*)
- At run time (as described in relation to the system variable `ConverseVar.printerAssociation`)

#### Related concepts

“EGL debugger” on page 369  
 “FormGroup part in EGL source format” on page 603  
 “Form part in EGL source format” on page 606  
 “Form part” on page 184  
 “Resource associations and file types” on page 393  
 “Segmentation in text applications” on page 189

#### Related reference

“pageEject()” on page 918  
 “printerAssociation” on page 1059

## Creating an EGL text form

A text form is an EGL form part that defines a layout to display in a command window. To create an EGL text form, do as follows:

1. Identify an EGL file to contain the text form and open that file in the EGL editor. You must create an EGL file if you do not already have one.
2. Type the specifics of the text form according to EGL syntax (for details, see *Form part in EGL source format*). You can use content assist to place an outline of the form part syntax in the file.
3. Save the EGL file.

#### Related concepts

“EGL projects, packages, and files” on page 15

"Form part" on page 184

"Text forms"

"Editing form groups with the EGL form editor" on page 195

### Related tasks

"Creating an EGL source file" on page 130

"Creating a form in the EGL form editor" on page 196

"Using the EGL templates with content assist" on page 131

### Related reference

"Content assist in EGL" on page 577

"Form part in EGL source format" on page 606

"Naming conventions" on page 778

## Text forms

Forms and their types are introduced in *Form part*. The current page outlines how to present text forms.

The **converse** statement is sufficient for giving the user access to a single, fixed text form. The logical flow of your program continues only after the user responds to the displayed form. You can also construct output from multiple forms, as in the following case:

- At the top of the output, a fixed form identifies a purchasing company and an order number
- In a subsequent floating area, a series of identically formatted floating forms identify each item of the company's order
- At the bottom of the output, a fixed form indicates the number of screens needed to scroll through the list of items

Two steps are necessary:

1. First, you construct the order-and-item output by coding a series of **display** statements, each of which adds a form to a runtime buffer but does not present data to the screen. Each **display** statement operates on one of the following forms:
  - Top form
  - Floating form, as presented by a **display** statement that is invoked repeatedly in a loop
  - Bottom form
2. Next, the EGL runtime presents all the buffered text forms to the output device in response to either of these situations:
  - The program runs a **converse** statement; or
  - The program ends.

In most cases, you present the last form of your screen output by coding a **converse** statement rather than a **display** statement.

The fixed forms each have an on-screen position, so the order in which you specify them, in relation to each other and in relation to the repeated display of floating forms, does not matter. The contents of the buffer are erased when output is sent to the screen.

If you overlay one text form with another, no error occurs, but the following statements apply:

- If a partial form overlays any lines in another fixed form, EGL replaces the existing form without clearing the rest of the output from the buffer. If you want to erase the existing output before displaying the new form, invoke the system function `ConverseLib.clearScreen` before issuing the **display** or **converse** statement for the new form.
- If you use a **display** or **converse** statement to place a floating map beyond the bottom of the floating area, all the floating forms in that floating area are erased, and the added form is placed on the first line of the same floating area.
- If a floating form overlays a fixed form, these statements apply- -
  - Only the fixed-form lines that are in the floating area are overwritten by the floating form
  - The result is unpredictable if a fixed-form line is overwritten by a floating-form line that includes a variable field

Whether you are presenting one form or many, the output destination is the screen device at which the user began the run unit.

#### Related concepts

“Form part” on page 184

#### Related reference

“Form part in EGL source format” on page 606

“FormGroup part in EGL source format” on page 603

“clearScreen()” on page 917

### Segmentation in text applications

Segmentation concerns how a program interacts with its environment before issuing a **converse** statement.

By default a program that presents text forms is *non-segmented*, which means that the program behaves as if it were always in memory and providing a service to only one user. The following rules are in effect before a non-segmented program issues a **converse** statement:

- Databases and other recoverable resources are not committed
- Locks are not released
- File and database positions are retained
- Single-user EGL tables are not refreshed; their values are the same before and after the converse
- Similarly, system variables are not refreshed

A called program is always non-segmented.

A non-segmented program can be easier to code. For example, you do not need to reacquire a lock on an SQL row after a **converse**. Disadvantages include the fact that SQL rows are held during user think time, a behavior that leads to performance problems for other users who need to access the same SQL row.

Two techniques are available for releasing or refreshing resources before a converse in a non-segmented program:

- You can set the system variable **ConverseVar.commitOnConverse** to 1. Results are as follows before a converse:
  - Databases and other recoverable resources are committed
  - Locks are released



- File and database positions are not retained, except when the database open statement includes the hold option, as is available only for COBOL programs
- The setting of **ConverseVar.commitOnConverse** never affects system variables or EGL tables.
- A second technique for handling converse is to set the *segmented* property of the text program to *yes*, either by changing a program property at development time or by setting the system variable **ConverseVar.segmentedMode** to 1 at run time. Segmentation causes the following results before a converse:
    - Databases and other recoverable resources are committed
    - Locks are released
    - File and database positions are not retained, even when the database open statement includes the hold option
    - Single-user EGL tables are refreshed; their values become the same as when the program began
    - System variables are refreshed; their values become the same as when the program began, except for a subset of variables whose values are saved *across segments*

The behavior of a segmented program is unaffected by the value of the system variable **ConverseVar.commitOnConverse**.

The benefit of using a segmented program on IMS or CICS for z/OS is as follows:

- In the absence of EGL, the developer of a segmented program writes code to analyze program state at each invocation. With EGL, the application logic is simpler because the developer writes code as if the user were having an ongoing conversation with a program that is always in memory.
- Limited resources such as memory are used more efficiently so that a larger number of terminals can run EGL programs at the same time.

The benefit of using a non-segmented program on IMS or CICS for z/OS is that the response time for each user is less than for a segmented program because program state is not saved and restored.

For additional details that are meaningful for IMS and CICS COBOL programs, see *Behavior of a segmented program on CICS and IMS*.

### Related concepts

“Behavior of a segmented program on CICS or IMS”

“Program part” on page 148

## Behavior of a segmented program on CICS or IMS

On CICS or IMS, an EGL segmented program acts as follows when the user first invokes it:

1. Performs initialization tasks, including a determination that the invocation is the user’s first
2. Gives control to the beginning of the program logic
3. Implements each EGL **converse** statement in this way--
  - a. Uses a work database to save *program state*, which is a set of user-specific values that reflect the current status of the user-program conversation. Included is the data for all records and forms. Also included is the information needed to run the program from the appropriate line.
  - b. Commits data base and recoverable resources.
  - c. Releases all locks.

- d. Does not retain database position, even if the database open statement includes the hold option.
- e. Converses a form.
- f. Ends.

When the user performs an action (to update business data, for example), the runtime system restores the program to memory. The program starts from the beginning again. The segmented program automatically acts as follows:

1. Performs initialization tasks, including a determination that the invocation is a continuation of processing for this user
2. Restores program state, including the data for all forms and records, as well as information about which **converse** statement ran in the program
3. Reads the user's input and performs any edits
4. Continues the cycle when implementing the next **converse** statement--
  - a. Saves program state
  - b. Commits database and recoverable resources
  - c. Releases all locks
  - d. Does not retain database position, even if the database open statement includes the hold option
  - e. Converses a form or VGUI record
  - f. Ends

#### Related concepts

"Segmentation in text applications" on page 189

#### Modified data tag and modified property

Each item on a text form has a *modified data tag*, which is a status value that indicates whether the user is considered to have changed the form item when the form was last presented.

As described later, an item's modified data tag is distinct from the item's **modified** property, which is set in the program and which pre-sets the value of the modified data tag.

**Interacting with the user:** In most cases, the modified data tag is pre-set to *no* when the program presents the form to the user; then, if the user changes the data in the form item, the modified data tag is set to *yes*, and your program logic can do as follows:

- Use a data table or function to validate the modified data (as occurs automatically when the modified data tag for the item is *yes*)
- Detect that the user modified the item (for example, by using a conditional statement of the type *if item modified*)

The user sets the modified data tag by typing a character in the item or by deleting a character. The modified data tag stays set, even if the user, before submitting the form, returns the field content to the value that was presented.

When a form is re-displayed due to an error, the form is still processing the same converse statement. As a result, any fields that were modified on the converse have the modified data tag set to *yes* when the form is re-displayed. For example, if data is entered into a field that has a validator function, the function can invoke the `ConverseLib.validationFailed` function to set an error message and cause the form

to re-display. In this case, when an action key is pressed, the validator function will execute again because the field's modified data tag is still set to *yes*.

**Setting the modified property:** You may want your program to do a task regardless of whether the user modified a particular field; for example:

- You may want to force the validation of a password field even if the user did not enter data into that field
- You may specify a validation function for a critical field (even for a protected field) so that the program always does a particular *cross-field validation*, which means that your program logic validates a group of fields and considers how one field's value affects the validity of another.

To handle the previous cases, you can set the **modified** property for a particular item either in your program logic or in the form declaration:

- In the logic that precedes the form presentation, include a statement of the type *set item modified*. The result is that when the form is presented, the modified data tag for the item is pre-set to *yes*.
- In the form declaration, set the **modified** property of the item to *yes*. In this case, the following rules apply:
  - When the form is presented for the first time, the modified data tag for the item is pre-set to *yes*.
  - If any of the following situations occurs before the form is presented, the modified data tag is pre-set to *yes* when the form is presented:
    - The code runs a statement of the type *set item initial*, which reassigns the original content and property values for the item; or
    - The code runs a statement of the type *set item initialAttributes*, which reassigns the original property values (but not content) for each item on the form; or
    - The code runs a statement of the type *set form initial*, which reassigns the original content and property values for each item on the form; or
    - The code runs a statement of the type *set form initialAttributes*, which reassigns the original property values (but not content) for each item on the form

The *set* statements affect the value of the **modified** property, not the current setting of the modified data tag. A test of the type *if item modified* is based on the modified data tag value that was in effect when the form data was last returned to your program. If you try to test the modified data tag for an item before your logic presents the form *for the first time*, an error occurs at run time.

If you need to detect whether the user (rather than the program) modified an item, make sure that the value of the modified data tag for the item is pre-set to *no*:

- If the **modified** property of the item is set to *no* in the form declaration, do not use a statement of the type *set item modified*. In the absence of that statement, the **modified** property is automatically set to *no* prior to each form presentation.
- If the **modified** property of the item is set to *yes* in the form declaration, use a statement of the type *set item normal* in the logic that precedes form presentation. That statement sets the **modified** property to *no* and (as a secondary result) presents the item as unprotected, with normal intensity.

**Testing whether the form is modified:** The form as a whole is considered to be modified if the modified data tag is set to *yes* for any of the variable form items. If you test the modified status of a form that was not yet presented to the user, the test result is FALSE.

**Examples:** Assume the following settings in the form *form01*:

- The **modified** property for the field *item01* is set to *no*
- The **modified** property for the field *item02* is set to *yes*

The following logic shows the result of various tests:

```
// tests false because a converse statement
// was not run for the form
if (form01 is modified)
;
end

// causes a runtime error because a converse
// statement was not run for the form
if (item01 is modified)
;
end

// assume that the user modifies both items
converse form01;

// tests true
if (item01 is modified)
;
end

// tests true
if (item02 is modified)
;
end

// sets the modified property to no
// at the next converse statement for the form
set item01 initialAttributes;

// sets the modified property to yes
// at the next converse statement for the form
set item02 initialAttributes;

// tests true
// (the previous set statement takes effect only
// at the next converse statement for the form
if (item01 is modified)
;
end

// assume that the user does not modify either item
converse form01;

// tests false because the program set the modified
// data tag to no, and the user entered no data
if (item01 is modified)
;
end

// tests true because the program set the modified
// data tag to yes
if (item02 is modified)
;
end
```

```

// assume that the user does not modify either item
converse form01;

// tests false
if (item01 is modified)
;
end

// tests false because the presentation was not
// the first, and the program did not reset the
// item properties to their initial values
if (item02 is modified)
;
end

```

---

## EGL form editor overview

The EGL form editor lets you edit a formGroup part graphically. The form editor works with formGroup parts, their form parts, and the fields in those form parts in much the same way as other graphical editors work with files like Web pages and Web diagrams.

The form editor has these parts:

- The editor itself, which displays the graphical representation of the form group and that form group's source code. You can switch between the graphical representation and the source code by clicking the **Design** and **Source** tabs at the bottom of the editor. Changes to the Source view or Design view are reflected immediately in the other view.
- The Properties view, which displays the EGL properties of the form or field currently selected in the editor.
- The Palette view, which displays the types of forms and fields that can be created in the editor.
- The Outline view, which displays a hierarchical view of the form group open in the editor.

For more information on using the form editor, see *Editing form groups with the EGL form editor*.

### Related concepts

- "FormGroup part" on page 183
- "Form part" on page 184
- "Display options for the EGL form editor" on page 204
- "Form filters in the EGL form editor" on page 206
- "Editing form groups with the EGL form editor" on page 195

### Related tasks

- "Creating a form in the EGL form editor" on page 196
- "Setting preferences for the EGL form editor" on page 204
- "Setting preferences for the EGL form editor palette entries" on page 200
- "Setting bidirectional text preferences for the EGL form editor" on page 205

---

## Editing form groups with the EGL form editor

The EGL form editor lets you edit a formGroup part graphically. The form editor works with formGroup parts, their form parts, and the fields in those form parts in much the same way as other graphical editors work with files like Web pages and Web diagrams. At any time, you can click the **Source** tab at the bottom of the editor and see the EGL source code that the editor is generating. The form editor has the following features:

- The form editor can edit the size and properties of a form group. To edit the properties of a form group, open the form group in the form editor and change its properties in the Properties view. To resize a form group, open it in the form editor and choose a size in characters from the list at the top of the editor.
- The form editor can create, edit, and delete forms in a form group. To create a form, click the appropriate type of form on the Palette view and draw a rectangle representing the size and location of the form in the editor. To edit a form, click it to select it, and then use the Properties view to edit its properties. You can also drag a form to move it, or resize it using the resize handles that appear on the border of a selected form. Many of the same options are available when you right-click a form to open its popup menu. See *Creating a form in the EGL form editor*.
- The form editor uses templates to create commonly used types of forms, such as popup forms and popup menus. These forms have pre-made borders, sections, and fields. See *Form templates in the EGL form editor*.
- The form editor can create, edit, and delete fields in a form. To create a field, click the appropriate type of field on the Palette view and draw a rectangle representing the size and location of the field in the editor. You can add a field only within an existing form. To edit a field, click it to select it, and then use the Properties view to edit its properties. You can also drag a field to move it, or resize it using the resize handles that appear on the border of a selected form. Many of the same options are available when you right-click a field to open its popup menu. See *Creating a constant field* or *Creating a variable field*.
- Filters can prevent forms from being shown in the form editor, allowing you to mimic the appearance of the form group at run time. To switch filters, create a filters, or edit filters, use the **Filters** button at the top of the editor. See *Form filters in the EGL form editor* or *Creating a filter*.
- You can customize the appearance of the form editor by using the display options at the top of the editor and by setting the editor's preferences in the Preferences window. For example, these options can display a grid over the form group, increase or decrease the zoom level, and show or hide sample values in fields. See *Display options for the EGL form editor* or *Setting preferences for the EGL form editor*.

### Related concepts

- “EGL form editor overview” on page 194
- “FormGroup part” on page 183
- “Form part” on page 184
- “Display options for the EGL form editor” on page 204
- “Form filters in the EGL form editor” on page 206
- “Form templates in the EGL form editor” on page 200

### Related tasks

- “Creating a filter” on page 196
- “Creating a popup form” on page 201
- “Creating a popup menu” on page 202

“Displaying a record in a text or print form” on page 202  
“Setting preferences for the EGL form editor” on page 204  
“Setting preferences for the EGL form editor palette entries” on page 200  
“Creating a form in the EGL form editor”  
“Creating a constant field” on page 197  
“Creating a variable field in a print or text form” on page 198

#### Related reference

“FormGroup part in EGL source format” on page 603  
“Form part in EGL source format” on page 606

## Creating a filter

To create a new filter in the EGL form editor, follow these steps:

1. Open a form group in the form editor.
2. In the form editor, click the **Filters** button. The Filters window opens.
3. In the Filters view, click the **New** button. The New Filter dialog opens.
4. In the New Filter dialog, type a name for the filter and click **OK**.
5. Select the forms to be displayed while the filter is active by doing one or more of the following steps:
  - Clear the check boxes next to the forms you want hidden by the filter.
  - Select the check boxes next to the forms you want shown by the filter.
  - Click the **Select All** button to show every form.
  - Click the **Deselect All** button to hide every form.
6. Click **OK**.

The new filter is now active. You can switch filters by using the list next to the **Filters** button.

#### Related concepts

“EGL form editor overview” on page 194  
“Editing form groups with the EGL form editor” on page 195  
“Display options for the EGL form editor” on page 204  
“Form filters in the EGL form editor” on page 206

#### Related tasks

“Creating a form in the EGL form editor”

## Creating a form in the EGL form editor

To create a form in the EGL form editor, follow these steps:

1. Open a form group in the form editor.
2. On the Palette view, click either **Text Form** or **Print Form**.
3. On the form group in the editor, click and drag a rectangle that indicates the size and shape of the form. The Create Form Part window opens.
4. In the Create Form Part Window, type a name for the form in the **Enter part name** field. This name will be the name of the form part in the EGL source code.
5. Click **OK**.
6. Click the form and edit its properties in the Properties view.
7. Add fields to the form as appropriate. See *Creating a constant field* and *Creating a variable field*.



You can also create forms based on the templates in the Palette view. These templates create forms with predefined appearances and fields. See *Creating a popup form* or *Creating a popup menu*.

#### Related concepts

- "EGL form editor overview" on page 194
- "Editing form groups with the EGL form editor" on page 195
- "FormGroup part" on page 183
- "Form part" on page 184
- "Display options for the EGL form editor" on page 204
- "Form filters in the EGL form editor" on page 206
- "Form templates in the EGL form editor" on page 200

#### Related tasks

- "Creating a filter" on page 196
- "Creating a popup form" on page 201
- "Creating a popup menu" on page 202
- "Displaying a record in a text or print form" on page 202
- "Creating a constant field"
- "Creating a variable field in a print or text form" on page 198

#### Related reference

- "FormGroup part in EGL source format" on page 603
- "Form part in EGL source format" on page 606

## Creating a constant field

Constant fields display a string of text that does not change in a form. Unlike variable fields, constant fields can not be accessed by EGL code. To insert a constant field into a form, follow these steps:

1. Open a form group in the EGL form editor.
2. If the form group has no forms, add a form to the form group. See *Creating a form*.
3. On the Palette view, click a type of constant field to add. The following types of constant fields are available by default:

Table 7. Constant fields available in the Palette view

Field name	Default color	Default intensity	Default highlighting	Default Protection
Title	Blue	Bold	None	Skip
Column Heading	Blue	Bold	None	Skip
Label	Cyan	Normal	None	Skip
Instructions	Cyan	Normal	None	Skip
Help	White	Normal	None	Skip

These fields are samples of commonly used constant text fields in a text-based interface. You can customize the individual fields after placing them on a form. You can also customize the default color, intensity, and highlighting of the fields available in the Palette view. See *Setting preferences for the EGL form editor palette entries*.

4. Within a form in the editor, click and hold the mouse to draw a rectangle that represents the size and location of the field. A preview box next to the mouse cursor shows you the size of the field and its location relative to the form.



**Note:** You can add a field only within an existing form.

5. When the field is the correct size, release the mouse. The new field is created.
6. Type the text you want to display in the field.
7. In the Properties view, set the properties for the new field.

#### Related concepts

“EGL form editor overview” on page 194

“Editing form groups with the EGL form editor” on page 195

“Form part” on page 184

#### Related tasks

“Setting preferences for the EGL form editor palette entries” on page 200

“Creating a variable field in a print or text form”

#### Related reference

“Form part in EGL source format” on page 606

## Creating a variable field in a print or text form

Variable fields can serve as input or output text in a form. Each variable field is based on an EGL primitive or a DataItem part. Unlike constant fields, variable fields can be accessed by EGL code. To insert a variable field into a form, follow these steps:

1. Open a form group in the EGL form editor.
2. If the form group has no forms, add a form to the form group. See *Creating a form*.
3. On the Palette view, click a type of variable field to add. The following types of variable fields are available by default:

Table 8. Variable fields available in the Palette view

Field name	Default color	Default intensity	Default highlighting	Default Protection
Input	Green	Normal	Underlined	No
Output	Green	Normal	None	Skip
Message	Red	Bold	None	Skip
Password	Green	Invisible	None	No

These fields are samples of commonly used variable text fields in a text-based interface. You can customize the individual fields after placing them on a form. You can also customize the default color, intensity, and highlighting of the fields available in the Palette view. See *Setting preferences for the EGL form editor palette entries*.

4. Within a form in the editor, click and hold the mouse to draw a rectangle that represents the size and location of the field. A preview box next to the mouse cursor shows you the size of the field and its location relative to the form.

**Note:** You can add a field only within an existing form.

5. When the field is the correct size, release the mouse. The New EGL Field window opens.
6. In the New EGL Field window, enter the name of the new field in the **Name** field.
7. Do one of the following to select the type of field:

- To use a primitive type, click a primitive type from the **Type** list.
- To use a DataItem part, follow these steps:
  - a. Click **dataItem** from the **Type** list. The Select a DataItem Part window opens.
  - b. In the Select a DataItem Part window, click a DataItem part from the list or type the name of one.
  - c. Click **OK**.
- 8. As necessary, type values in the **Dimensions** field or fields to set the dimensions of the new variable field.
- 9. If you want to make the field an array, select the **Array** check box.
- 10. If the **Array** check box is selected, click **Next** and continue following these steps. Otherwise, click **Finish** and stop following these steps. The new field is created and you do not need to follow the rest of these steps, because they are applicable only if you are creating an array.
- 11. On the Array Properties page of the New EGL Field window, type the size of the array in the **Array Size** field.
- 12. Choose an orientation of **Down** or **Across** from the **Index Orientation** buttons.
- 13. Under **Layout**, enter the number of vertical and horizontal fields in the **Fields Down** and **Fields Across** fields.
- 14. Under **Spaces**, enter the amount of space between the array's rows and columns in the **Lines between rows** and **Spaces between columns** fields.
- 15. Click **Finish**. The new field is created in the form group.

Once you have created the new field, click the field to select it and set the properties for the field in the Properties view.

Since variable fields have no default value, they can be invisible if they are not highlighted. To mark each variable field with appropriate sample text, click the **Toggle Sample Values** button at the top of the editor.

Once you have created a variable field, you can double-click it in the editor to open the Edit Type Properties window. From this window you can edit the field in the following ways:

- Change the field's name by typing a new name in the **Field Name** field.
- Select a new type of field from the **Type** list.
- Change the precision of the field by entering a new number in the **Precision** field.

When you are finished editing the field's properties in the Edit Type Properties window, click **OK**.

### Related concepts

"EGL form editor overview" on page 194

"Editing form groups with the EGL form editor" on page 195

"Form part" on page 184

### Related tasks

"Setting preferences for the EGL form editor palette entries" on page 200

"Creating a constant field" on page 197

#### Related reference

“Form part in EGL source format” on page 606

## Setting preferences for the EGL form editor palette entries

The preferences for the EGL form editor palette control the default color, intensity, and highlighting for the types of constant and variable fields in the palette. To change these preferences, follow these steps:

1. From the menu bar, click **Window > Preferences**. The Preferences window opens.
2. In the left pane of the Preferences window, expand **EGL > EGL Form Editor** and click **EGL Palette Entries**.
3. In the right pane of the Preferences window, for each type of constant and variable field in the **Palette Entries** list, select the following options:
  - From the **Color** list, click a default color for that type of field.
  - From the **Intensity** list, click a default intensity for that type of field.
  - From the **Highlight** radio buttons, click a default highlighting style for that type of field.
  - From the **Protect** radio buttons, choose whether the field is protected from user update by default. For more information about protecting fields, see *ConsoleField properties and fields*.

**Note:** You can restore all of the palette entries to their default settings by clicking **Restore Defaults**.

4. When you are finished setting the preferences for the palette entries, click **OK**

#### Related concepts

“EGL form editor overview” on page 194

“Editing form groups with the EGL form editor” on page 195

#### Related tasks

“Setting preferences for the EGL form editor” on page 204

“Creating a constant field” on page 197

“Creating a variable field in a print or text form” on page 198

#### Related reference

“ConsoleField properties and fields” on page 533

## Form templates in the EGL form editor

The EGL form editor uses templates to create commonly used types of forms and fields. These templates are listed in the **Templates** drawer of the Palette view.

The form editor can create forms from templates. These forms have pre-made borders, sections, and fields. To create a form from a template, see *Creating a popup form* or *Creating a popup menu*.

The form editor can create a group of fields using an EGL record as a template. To create fields representing data from an EGL record, see *Displaying a record in a form*.

#### Related concepts

“EGL form editor overview” on page 194

“Editing form groups with the EGL form editor” on page 195

### Related tasks

"Creating a popup form"

"Creating a popup menu" on page 202

"Displaying a record in a text or print form" on page 202

"Setting preferences for the EGL form editor" on page 204

"Setting preferences for the EGL form editor palette entries" on page 200

"Creating a form in the EGL form editor" on page 196

### Creating a popup form

A popup form is a special kind of form that can be added to a form group. Fundamentally, a popup form is the same as an ordinary text form, but popup forms are created with pre-made features like borders and sections. To create a popup form in the EGL form editor, follow these steps:

1. Open a form group in the form editor.
2. On the Palette view, click **Popup Form**.
3. On the form group in the editor, click and drag a rectangle that indicates the size and shape of the popup form. The Create Form Part window opens.
4. In the Create Form Part Window, type a name for the form in the **Enter part name** field. This name will be the name of the form part in the EGL source code.
5. Click **OK**. The New Popup Form Template window opens.
6. In the New Popup Form Template window, enter the characters to use for the form's borders in the **Vertical Character** and **Horizontal Character** fields.
7. Click a color for the border from the **Color** list.
8. Click an intensity for the border from the **Intensity** list.
9. Click a highlight value from the **Highlight** radio buttons.
10. Repeat the following steps for each section you want to add to the form. You must add at least one section to the form.
  - a. Under **Popup Sections**, click the **Add** button. The Create Popup Form Section window opens.
  - b. In the Create Popup Form Section window, type a name for the section in the **Section Name** field.
  - c. In the **Number of rows** field, enter the number of rows in the section. Do not enter a number greater than the number of remaining effective rows, which is displayed at the bottom of the New Popup Form Template window.
  - d. Click **OK**.
  - e. Use the **Up** and **Down** buttons to set the order of the fields.

**Note:** The total number of rows in the popup field's sections cannot exceed the total number of rows in the popup field. When adding sections, pay attention to the **Remaining Effective Rows** field and remember that dividers between the sections require an additional row for each new field.

11. When you are finished adding sections to the popup field, click **Finish**. The new popup form is created in the editor.
12. Add fields to the form as appropriate. See *Creating a constant field* and *Creating a variable field*.

### Related concepts

"EGL form editor overview" on page 194

“Editing form groups with the EGL form editor” on page 195  
“Form templates in the EGL form editor” on page 200

#### Related tasks

“Setting preferences for the EGL form editor” on page 204  
“Creating a form in the EGL form editor” on page 196  
“Creating a constant field” on page 197  
“Creating a variable field in a print or text form” on page 198  
“Creating a popup menu”

### Creating a popup menu

A popup menu is a special kind of form that can be added to a form group. Fundamentally, a popup menu is the same as an ordinary text form, but popup menus are created with pre-made features like a title, help text, and a specified number of menu options. To create a popup menu in the EGL form editor, follow these steps:

1. Open a form group in the form editor.
2. On the Palette view, click **Popup Menu**.
3. On the form group in the editor, click and drag a rectangle that indicates the size and shape of the popup menu. The Create Form Part window opens.
4. In the Create Form Part Window, type a name for the form in the **Enter part name** field. This name will be the name of the form part in the EGL source code.
5. Click **OK**. The New Popup Menu Template window opens.
6. In the New Popup Menu Template, enter the size of the popup menu in the **Width** and **Height** fields. By default, these fields are populated with the size of the form you created in the editor.
7. In the **Menu Title** field, type the title of the menu.
8. In the **Number of Menu Options** field, type the number of menu options that the popup menu will have.
9. In the **Menu Help Text** field, type any additional help text for the menu.
10. Click **Finish**. The new popup menu is created in the editor.
11. Add fields to the new popup menu and edit the existing fields as appropriate. See *Creating a constant field* and *Creating a variable field*.

#### Related concepts

“EGL form editor overview” on page 194  
“Editing form groups with the EGL form editor” on page 195  
“Form templates in the EGL form editor” on page 200

#### Related tasks

“Setting preferences for the EGL form editor” on page 204  
“Creating a form in the EGL form editor” on page 196  
“Creating a constant field” on page 197  
“Creating a variable field in a print or text form” on page 198  
“Creating a popup form” on page 201

### Displaying a record in a text or print form

The **Record** template, found in the **Templates** drawer of the Palette view, creates a group of form fields that are equivalent to the fields in an EGL record part. To create the form fields, follow these steps:

1. Open a form group in the EGL form editor.
2. Create a form. See *Creating a form in the EGL form editor*.

3. On the Palette view, click **Record**.
4. Within a form in the editor, click and hold the mouse to draw a rectangle that represents the size and location of the fields. A preview box next to the mouse cursor shows you the size of the record fields and their location relative to the field.

**Note:** You can add a record only within an existing form.

5. When the record is the correct size, release the mouse. The EGL Record Placement window opens.
6. In the EGL Record Placement, click **Browse**. The Select a Record Part dialog opens.
7. In the Select a Record Part dialog, click the name of the record part you want to use or type the name of a record part.
8. Click **OK**. Now the Create a Record window is populated with a list of the fields in that record.
9. Using one or more of the following methods, select and organize the record part fields you want to display as fields in the form:
  - To remove a field, click its name and then click **Remove**.
  - To add a field, follow these steps:
    - a. Click the **Add** button. The Edit Table Entry window opens.
    - b. In the Edit Table Entry window, type a name for the field in the **Field Name** box.
    - c. In the **Type** list, select a type for the field.
    - d. If necessary for the type you have selected, enter the precision for the field in the **Precision** field.
    - e. Enter a width for the field in the **Field Width** field.
    - f. If you want the field to be an input field, select the **Make this field an input field** check box. Otherwise, clear the check box.
    - g. Click **OK**.
  - To edit a field, follow these steps:
    - a. Click the field's name.
    - b. Click the **Edit** button. The Edit Table Entry window opens.
    - c. In the Edit Table Entry window, type a name for the field in the **Field Name** box.
    - d. In the **Type** list, select a type for the field.
    - e. If necessary for the type you have selected, enter the precision for the field in the **Precision** field.
    - f. Enter a width for the field in the **Field Width** field.
    - g. If you want the field to be an input field, select the **Make this field an input field** check box. Otherwise, clear the check box.
    - h. Click **OK**.
  - To move fields up or down in the list, use the **Up** and **Down** buttons.
10. Using the **Orientation** radio buttons, choose a vertical or horizontal orientation for the fields.
11. In the **Number of Rows** field, enter the number of rows you want the group of fields to have.
12. If you want the group of fields to have a header row, select the **Create header row** check box.
13. Click **Finish**.

### Related concepts

"EGL form editor overview" on page 194

"Editing form groups with the EGL form editor" on page 195

"Form templates in the EGL form editor" on page 200

### Related tasks

"Creating a form in the EGL form editor" on page 196

"Creating a constant field" on page 197

"Creating a variable field in a print or text form" on page 198

## Display options for the EGL form editor

The EGL form editor has display options that allow you to control how form groups appear in the editor at design time. These options do not change the appearance of the form group at run time. From left to right at the top of the editor, these are the buttons that control the display options:

### Toggle Gridlines

This option displays a grid over the form group to help in sizing and arranging forms. To change the color of the grid, see *Setting preferences for the EGL form editor*.

### Toggle Sample Values

This option inserts sample values into variable fields, which would otherwise be invisible.

### Toggle Black and White Mode

This option switches the editor's background from black to white.

### Zoom Level

Sets the magnification level of the editor.

There are other buttons at the top of the editor that control the size of the form group and the editor's filters. See *Editing form groups with the EGL form editor* or *Form filters in the EGL form editor*.

### Related concepts

"EGL form editor overview" on page 194

"Form filters in the EGL form editor" on page 206

### Related tasks

"Editing form groups with the EGL form editor" on page 195

"Creating a filter" on page 196

"Setting preferences for the EGL form editor"

## Setting preferences for the EGL form editor

The preferences for the EGL form editor can change the appearance of the form editor, such as the background color and grid color. To change the preferences for the form editor, follow these steps:

1. From the menu bar, click **Window > Preferences**. The Preferences window opens.
2. In the left pane of the Preferences window, expand **EGL** and click **EGL Form Editor**.
3. In the right pane of the Preferences window, select the preferences for the form editor:
  - In the **Background Color** field, select a background color for the form editor.
  - In the **Grid Color** field, select a grid color for the form editor.



- If you want to show a border around fields, select the **Highlight fields** check box and select a color.
- If you want to show rulers at the top and left side of the form editor, select the **Show rulers** check box.
- In the **Font** list, click a font for the fields and click a size from the adjacent list.

**Note:** Choose a monospaced font to ensure that your fields display at the correct size in the form editor. A monospaced font is a font whose characters all have the same width, such as Courier New.

- If you want blinking fields to be displayed in italic type in the editor, select the **Visually demonstrate blinking fields** check box. This option does not change the appearance of the fields at run time; it only changes their appearance at design time.

**Note:** You can restore the EGL form editor preferences window to its default settings by clicking **Restore Defaults**.

4. When you are finished setting the preferences for the EGL form editor, click **OK**.

#### Related concepts

"EGL form editor overview" on page 194

"Editing form groups with the EGL form editor" on page 195

"Form filters in the EGL form editor" on page 206

#### Related tasks

"Setting preferences for the EGL form editor palette entries" on page 200

"Setting bidirectional text preferences for the EGL form editor"

## Setting bidirectional text preferences for the EGL form editor

The EGL form editor supports languages such as Arabic and Hebrew in which the text is written right to left but numbers and Latin alphabetic strings within the text are presented left to right. For more information, see *Bidirectional language text*. To change the preferences for the form editor's support of bidirectional languages, follow these steps:

1. From the menu bar, click **Window > Preferences**. The Preferences window opens.
2. In the left pane of the Preferences window, expand **EGL > EGL Form Editor** and click **EGL bidi preferences**.
3. In the right pane of the Preferences window, select the preferences for the form editor's support of bidirectional languages:
  - To enable the use of bidirectional language options in the form editor, select the **Bi-directional options enabled** check box. Once this check box is selected, all of the other options are available.
  - To display the bi-directional text fields in the way they will appear, with the correct ordering of letters and numbers (visual ordering), instead of in the way the characters are typed (logical ordering), select the **Enable visual ordering** check box.
  - To place a button on the form editor that allows you to change the bidirectional settings for a formGroup, select the **Enable bi-directional settings button** check box. With this button, you can change the bidirectional settings for an individual formGroup to be different from the settings in the Preferences window.



- To use right to left map coordinates for the form editor, select the **Default RTL orientation** check box. When a form group is set to use right to left map coordinates, the top right corner is defined as the location (1,1). With a left to right orientation, the location (1,1) is the top left corner.
- To reverse directional punctuation characters like < and (, select the **Enable symmetric swapping** check box.
- To switch numerals from Hindi to Arabic, select the **Enable numeric swapping** check box.

**Note:** You can restore the EGL form editor preferences window to its default settings by clicking **Restore Defaults**.

#### Related concepts

“EGL form editor overview” on page 194

“Editing form groups with the EGL form editor” on page 195

“Form filters in the EGL form editor”

#### Related tasks

“Setting preferences for the EGL form editor palette entries” on page 200

#### Related reference

“Bidirectional language text” on page 561

## Form filters in the EGL form editor

Filters limit which forms are shown in the EGL form editor. You can define any number of filters, but only one filter can be active at a time. Filters affect only the presentation of the form group at design time; they do not affect the EGL code in any way. See *Creating a filter*.

You can switch between active filters with the list next to the **Filters** button at the top of the editor. To create, edit, or delete filters, click the **Filters** button.

From the Filters window, you can manage your filters by using the following functions:

- Select filters from the list.
- Add a new filter by clicking **New**.
- Delete a filter by selecting it from the list and clicking **Remove**.
- Select which forms are displayed when the filter is active.

#### Related concepts

“EGL form editor overview” on page 194

“Editing form groups with the EGL form editor” on page 195

#### Related tasks

“Creating a filter” on page 196

“Creating a form in the EGL form editor” on page 196

---

# Creating a user interface with ConsoleUI

---

## Console user interface

The console user interface (ConsoleUI) is a technology for displaying data in a text-based format on a Windows or UNIX screen. This technology is available only in EGL-generated Java programs, not in PageHandlers.

The interface that you create with ConsoleUI can be displayed in Windows 2000/NT/XP or UNIX X-windows, either locally or by way of a remote terminal session.

ConsoleUI is distinct from Text user interface (TextUI), and the two cannot operate in the same program:

- When TextUI is in effect, the style of interface is like that used in a mainframe program interacting with 3270 terminals. The program presents a text form but does not process user input as the user moves from one field to the next. When the user submits the form (by pressing the **Enter** key, in most cases), all the data in the form returns to the program, and only then does the program validate the data; if validation succeeds, the program runs the next coded statement.
- When ConsoleUI is in effect, the style of interface is like that used in a UNIX-based program interacting with character-based terminals. The program presents a console form and can respond immediately to a user event, as when the user presses the **Tab** key to move an on-screen cursor to the next field. Validation is on a field-by-field basis, and you can restrict the cursor to the current field until the user has typed valid data there.

When you use consoleUI, you typically code a program as follows:

1. Declare a set of variables that are based on the ConsoleUI parts, which are always available; you do not define the parts that are specific to ConsoleUI.
2. Open a visual entity such as a form by including a consoleUI variable as an argument when you invoke the appropriate EGL function. Alternatively, you can open a visual entity by invoking an EGL function like **displayFormByName**, which accepts a name that is known at run time.
3. Reference the visual entity in an EGL **openUI** statement, which allows for user interaction by tying particular events (such as user keystrokes) to particular logic.

The user of a consoleUI application can press keys to interact with the on-screen display, but mouse clicks have no effect.

ConsoleUI can accept user input into a field, but only if you have specified a *binding*, which is a correspondence between the input field and a variable of primitive type. The EGL runtime acts as follows:

- Uses the variable value as the initial content of a displayed field; and
- Moves the user's input to that variable as soon as the user leaves the field.

ConsoleUI also allows you to interact with users in *line mode*, which is a mode of processing in which your code reads or writes only one line at a time. The implications of line mode are as follows:

- In the Eclipse workbench, the user interacts with the Console view
- In a program that was invoked with a command prompt, the user interacts with the command window
- In a program that runs under Curses in UNIX, the user interacts with the window in which the UI is displayed; and the usual window-based interaction is suspended

ConsoleUI is equivalent to the user-interface technology in the Informix 4GL product.

#### Related tasks

"Creating an interface with ConsoleUI"

#### Related reference

"ConsoleUI parts and related variables" on page 209

"ConsoleUI screen options for UNIX" on page 213

"EGL library ConsoleLib" on page 886

"openUI" on page 726

"Use of new in ConsoleUI" on page 212

---

## Creating an interface with ConsoleUI

Use the following steps to create an interface with ConsoleUI:

1. Create an EGL source file
2. Write a program that includes the language elements described in *ConsoleUI parts and related variables*
3. Generate Java code from the EGL source file
4. Run the generated Java file as an application

Each of these tasks is detailed below.

#### Creating an EGL source file

1. In the workbench, from the EGL Perspective, select **File>New>EGL Source File**. Or, from any perspective, select **File>New>Other>EGL Source File..**
2. In the wizard screen, enter the following information:
  - **Source Folder:** the directory location that will contain the EGL source file.
  - **Package:** the package location that will contain the EGL source file. This field is optional.
  - **EGL Source File Name:** the filename of the Console UI source file, such as **myConsoleUI**.
3. Select **Finish** to create the file. An extension (**.egl**) is automatically appended to the end of the file name. The EGL source file appears in the Project Explorer view and automatically opens in the default EGL editor.

#### Writing the ConsoleUI program

To populate the source file and create the interface, you will need ConsoleUI variables and functions. to use the ConsoleUI language elements, which are introduced in the overview help topic, and defined thoroughly in the individual ConsoleUI **Library**, **OpenUI Statement**, **Record types**, and **Enumerations** help topics.

A ConsoleUI application must include, at a minimum, the following elements:

1. PROGRAM...END
2. Function main ()
3. OpenUI Statement

**Note:** Although the **OpenUI statement** is fundamental to the ConsoleUI, you can write a successful ConsoleUI program without an **OpenUI statement**.

### Generating Java code from EGL source

To generate a Java file:

1. In the EGL Editor, right-click on the ConsoleUI file. A context menu displays.
2. Select **Generate**.

**Note:** A ConsoleUI .egl source file cannot be generated to COBOL.

### Run the generated Java file as an application

To run the generated Java file:

1. From the Project Explorer, right-click on the generated Java (.java) file. A context menu displays.
2. Select **Run>Run As>Java Application**.
3. Or, with the Java file open in the editor, select **Run>Run As>Java Application** from the main menu.
4. Your ConsoleUI will display to a window.

A ConsoleUI application can display in either a curses-based terminal session or a Swing-based graphical window. UNIX users have a more flexible display choice, which is described in the *ConsoleUI screen options for UNIX* help topic.

**Note:** IBM

### Related concepts

"Console user interface" on page 207

### Related reference

"EGL library ConsoleLib" on page 886

"ConsoleUI parts and related variables"

"ConsoleUI screen options for UNIX" on page 213

"openUI" on page 726

---

## ConsoleUI parts and related variables

When you work with consoleUI, you create the following kinds of variables, which are based on the related consoleUI parts:

- Window
- Prompt
- ConsoleField
- ConsoleForm
- Menu
- MenuItem

The library **ConsoleLib** also includes system variables of type `PresentationAttributes`. The system variables control visual aspects of displayed output. To change aspects of your display, you can change those variables by setting the `PresentationAttributes` fields **color**, **highlight**, and **intensity**. For details on those fields, see *PresentationAttributes fields in EGL consoleUI*.

## Window

A window is a rectangular area in which you can place other visual entities that are represented as variables.

When you display a window and no other windows are in effect, the new window is inside the *screen window*, which is a rectangle that has the basic characteristics of any window in the operating system. This is not the case in UNIX when the Curses library is in use; there the display of a consoleUI window puts the existing terminal window into windowing mode.

Any additional window that you display appears in the content portion of the screen window, usually on top of the window that you already opened. A side-by-side presentation of windows is also possible.

When you declare a window, you can set various properties. **Position**, for example, is the location relative to the upper left corner of the display; and **size** is the window's height and width in number of characters. You may use variables as well as literal values to specify both size and position.

An example of a window declaration is as follows:

```
myWindow WINDOW
{name="myWindow", position = [2,2],
 size = [18,75], color = red, hasborder=yes};
```

You display a window by using an EGL function whose name begins with *ConsoleLib.openWindow*. If you have not displayed a window when presenting other data, EGL provides a window for you.

## Prompt

A prompt is a one-line statement that elicits user input. A declaration of a prompt is as follows:

```
myPrompt Prompt { message = "Type your ID: "};
```

You display a prompt by including the variable in an **openUI** statement, which binds the prompt to a variable of type `String`, but only for input. You can configure the prompt to accept a single character or a string.

## ConsoleField

A `consoleField` is an on-screen field that is declared in the context of a console form (as described later). The next example declares a `consoleField` whose content can vary at run time:

```
myField ConsoleField (
    name="myFieldName",
    position=[1,31],
    fieldLen=20,
    binding = "myVariable" );
```

To specify constant text, use an asterisk (\*) in place of the variable name, as in the following example:

```
* ConsoleField
{ position=[2,5], value="Title: " };
```

It is highly recommended that when you declare a named consoleField, you use the same name for the consoleField and for the value of the name attribute within the consoleField. However, different names are valid for those two uses. You would reference the consoleField name (like *myField*) when access to the consoleField is resolved at generation time. You would reference the name-attribute value (like *myFieldName*) when access is resolved at run time, as when the consoleField is used to define an event in the **openUI** statement.

## ConsoleForm

A consoleForm is primarily a set of consoleFields. To make a consoleForm active, you invoke the system function **ConsoleLib.displayForm**. To display a read-only consoleForm, for example, you can do as follows:

1. Invoke **ConsoleLib.displayForm**
2. Invoke the system function **ConsoleLib.getKey** to wait for a user keystroke

To allow the user to write to a consoleField, do this instead:

1. Invoke **ConsoleLib.displayForm**
2. Issue an **openUI** statement that references either the displayed consoleForm or specific consoleFields in the consoleForm.

The consoleForm is a record of subtype ConsoleForm and can include not only consoleFields, but any of the fields that are valid in any EGL record.

To allow for user interaction with an on-screen table of consoleFields, do this:

1. In the consoleForm, declare an arrayDictionary that in turn references consoleField arrays that are also declared in the consoleForm
2. Use that arrayDictionary in an **openUI** statement

To allow for user interaction with only a subset of consoleFields in the consoleForm, you can list the consoleFields in the **openUI** statement, either in explicitly or by referencing a dictionary. Like the arrayDictionary, the dictionary is declared in the consoleForm and references consoleFields that are also declared in the consoleForm.

EGL does not display any primitive variable that you declare in the consoleForm. You can use such a variable to bind a consoleField, as you can use a variable declared outside of the consoleForm.

In general, you create consoleForm bindings in either of two ways:

- By setting a default binding when you declare the consoleForm.
- By setting a binding when you code the **openUI** statement.

Any binding specified in the **openUI** statement overrides the default binding in total; none of the consoleForm-declaration bindings remain.

If you use the **openUI** statement to bind variables, one option is to use the statement property **isConstruct**, which acts as follows:

- Formats user input into a string appropriate to an SQL WHERE clause

- Places that string into a single variable so you can easily code an SQL SELECT statement that retrieves user-requested data from a relational database, as when you code an EGL **prepare** statement

For details on the property **isConstruct**, see *OpenUI statement*.

*Tab order* is the order in which the user tabs from one consoleField to another. By default, the tab order is the order of the consoleFields in the consoleForm declaration. If you provide a list of consoleFields in an **openUI** statement, the tab order is the order of consoleFields in that statement; similarly, if you provide a dictionary or arrayDictionary in an **openUI** statement, the tab order is the order of consoleFields in the declaration of the dictionary or arrayDictionary.

By default, the user exits a consoleForm-related **openUI** statement by pressing the **Esc** key.

## Menu

A menu is a set of labels displayed horizontally. One label is for the menu as a whole and one for each menuItem in the menu. To ensure that a response occurs when the user selects a particular menuItem, you reference the menu as a whole in the **openUI** statement and reference the menuItem in an OnEvent clause of that statement.

## MenuItem

A menuItem displays a label and is used as described in the previous section.

### Related concepts

“ArrayDictionary” on page 87  
 “Console user interface” on page 207  
 “Dictionary” on page 83

### Related reference

“ConsoleField properties and fields” on page 533  
 “ConsoleForm properties in EGL consoleUI” on page 546  
 “EGL library ConsoleLib” on page 886  
 “ConsoleUI screen options for UNIX” on page 213  
 “Menu fields in EGL consoleUI” on page 547  
 “MenuItem fields in EGL consoleUI” on page 548  
 “openUI” on page 726  
 “PresentationAttributes fields in EGL consoleUI” on page 550  
 “Prompt fields in EGL consoleUI” on page 551  
 “Window fields in EGL consoleUI” on page 553

### Related tasks

“Creating an interface with ConsoleUI” on page 208

---

## Use of new in ConsoleUI

When you create an EGL program that uses consoleUI, every variable of type Menu, MenuItem, Prompt, and Window is a *reference variable*, which contains a memory address that refers to a value stored outside the variable.

You can declare a reference variable as you declare any other, as in this example:

```
myPrompt Prompt { message = "Type your ID: "};
```

Alternatively, you can declare a reference variable and initialize it with the reserved word **new**, as in this example:

```
myPrompt Prompt = new Prompt { message = "Type your ID: "};
```

When you are declaring variables, the difference between the two formats has little practical effect; but when you code the `openUI` statement, the word **new** provides a coding convenience, as shown in *openUI*.

The general syntax for **new** is as follows:

```
new partName
```

*partName*

One of the following words, which refers to a particular kind of part:

- Menu
- MenuItem
- Prompt
- Window

For details on other implications of reference variables, see *Reference compatibility in EGL*.

#### **Related concepts**

“Console user interface” on page 207

#### **Related reference**

“ConsoleUI parts and related variables” on page 209

“openUI” on page 726

“Reference compatibility in EGL” on page 862

#### **Related tasks**

“Creating an interface with ConsoleUI” on page 208

---

## **ConsoleUI screen options for UNIX**

EGL users on the supported UNIX platforms have the ability to run their ConsoleUI application using either a graphical display mode or a UNIX curses mode.

### **Graphical display mode**

To run a ConsoleUI application in graphical display mode, you must make sure that the EGL curses library is not located in the Library Path environment variable of your running shell. This is the default mode.

### **UNIX curses mode**

To run a ConsoleUI application in UNIX curses mode, you must have the appropriate platform-specific EGL curses library in the Library Path environment variable of your running shell. The EGL curses libraries must be downloaded from the EGL Support website.

#### **To download the EGL curses library:**

1. Locate the appropriate EGL Support website.
  - The URL for Rational Application Developer is:



<http://www3.software.ibm.com/ibmdl/pub/software/rational/sdp/rad/60/redist>

- The URL for Rational Web Developer is:

<http://www3.software.ibm.com/ibmdl/pub/software/rational/sdp/rwd/60/redist>

2. Download the **EGLRuntimesV60IFix001.zip** file to your preferred directory.

3. Unzip **EGLRuntimesV60IFix001.zip** to identify the following files:

- AIX: **EGLRuntimes/Aix/bin/libCursesCanvas6.so**
- Linux: **EGLRuntimes/Linux/bin/libCursesCanvas6.so**

4. Insert the appropriate EGL curses library in the Library Path environment variable.

**AIX:** Set the '**LIBPATH**' Library Path environment variable using the following bourne-shell:

```
"LIBPATH=$INSTDIR/aix; export LIBPATH"
```

**Linux:** Set the '**LD\_LIBRARY\_PATH**' Library Path environment variable using the following bourne-shell:

```
"LD_LIBRARY_PATH=$INSTDIR/aix; export LD_LIBRARY_PATH"
```

#### **Related concepts**

"Console user interface" on page 207

#### **Related reference**

"EGL library ConsoleLib" on page 886

"ConsoleUI parts and related variables" on page 209

"openUI" on page 726

#### **Related tasks**

"Creating an interface with ConsoleUI" on page 208

---

## Creating an EGL Web application

---

### Web support

EGL provides support for Web-based applications in the following ways:

- You can develop a *PageHandler*, which is a logic part whose functions are each invoked by a specific user action at a Web page. Generation of this logic part also can provide a JavaServer Faces JSP for customization.
- You can provide functionality that is common to several Web pages, as when each page in an application displays a button that the user clicks to log out. The user's click in this case can invoke an EGL program, which acts as a common subroutine.
- You may want to retrieve an item of information (such as a stock price) or cause another action (such as building an output string to send by e-mail) when the functionality is useful across multiple applications. In this case, build a Web service, which is a set of operations that can be invoked by many Internet-based clients. For an overview, see *EGL services and Web services*.
- You can update a *VGWebTransaction* program, which allows you to structure a Web application as if you were developing a text application. This type of program is supported for VisualAge Generator migration and is not recommended for new development. For an overview, see *Use of an EGL program in a Web application*.
- Finally, when you work in the WebSphere Page Designer, you can customize JavaServer Faces JSPs and can affect *PageHandlers*, as described in *Page Designer Support for EGL*.

#### Related concepts

"EGL services and Web services" on page 158

"PageHandler" on page 223

"Use of an EGL program in a Web application"

"WebSphere Application Server and EGL" on page 425

#### Related tasks

"Starting a Web application on the local machine" on page 424

#### Related reference

"Page Designer support for EGL" on page 221

---

## Use of an EGL program in a Web application

An EGL program can participate in a Web application in several ways:

- A *VGWebTransaction* program can exchange data between the fields on the Web page and the fields in a *VGUI record*. The program controls the overall flow of events, responding to user data and to the user event that caused the submission of data.
- A *VGWebTransaction* program can forward control to a *PageHandler*, in which case subsequent processing is controlled by the combination of page and *PageHandler*, not by the program. Similarly, an EGL *PageHandler* can forward control to a *VGWebTransaction* program; and either part can forward control to a part of the same kind.

- An EGL called program can be invoked from a PageHandler or from a VGWeb Transaction program.

**Note:** The VGWebTransaction program is provided primarily to support migration from VisualAge Generator. It is recommended that you use PageHandlers for new development.

## Programmatic control and the VGUI record

When you present a Web page by displaying a VGUI record from a VGWebTransaction program, you can ensure the following sequence:

1. The user submits page data
2. Processing continues with the next statement in the current program or with the first statement in a specified program

By repeatedly forcing the invocation of specific logic, you can cause a sequence of Web pages to be presented in the same browser window.

To ensure that a specific program is invoked after the user submits data from a Web page, the originating program uses one of two kinds of EGL statements to present the page:

- The **converse** statement; or
- The **show** statement with a returning clause.

Alternatively, to let the user rather than the program determine which program is invoked in response to a Web page, present the page with a **show** statement that has no returning clause.

However you create a Web application, you can include buttons and hypertext links to allow the user to display new Web pages in *different* (new) browser windows.

### converse statement

The **converse** statement ensures that, after the user submits data, the next statement in the same program is invoked. Also, the variable values in the program (as stored in a work database) are the same as when the page was presented, with these exceptions:

- Changes made to the VGUI record are in effect when the program regains control.
- The values of some system variables are lost. For details on a specific variable, see the help page for that variable, as listed in *System words in alphabetical order*.

A program design that uses a **converse** statement is relatively simple; but if you are generating a Java program, you get better performance by using a **show** statement to return to the beginning of the same program. Use of a **show** statement requires a more complicated design, however, because the re-invoked program starts at the first line, and that initial code must analyze whether the program is being invoked at the beginning or in the middle of a user-code interaction.

For other details on the **converse** statement, see *converse*.

### show statement

The **show** statement with a returning clause ensures that, after the user submits data, the program specified in the returning clause is invoked. You can code the

statement so that the invoked program receives data directly from the originating program; in this way you can retain variable values for use at a later stage of the user-code interaction.

If you present a Web page by using a **show** statement *without* a returning clause, the Web page is presented without directing subsequent processing. In this case, you can include buttons and hypertext links to give the user a range of choices for what is displayed in the same (or a different) browser window.

For other details, see *show*.

#### **Related concepts**

“VGUIRecord part” on page 167

#### **Related reference**

“converse” on page 672

“Page Designer support for EGL” on page 221

“show” on page 751

---

## **Creating a single-table EGL Web application**

### **EGL Data Parts and Pages wizard**

The EGL Data Parts and Pages wizard gives you a convenient way to create a Web-based utility that lets you maintain a specific table in a relational database.

The wizard creates these entities:

- A set of PageHandlers that you later generate into a set of parts that run under Java Server Faces
- An SQL record part, as well as the related dataItem parts and library-based function parts
- A set of JSP files that provide the following Web pages:
  - A *selection condition page*, which accepts selection criteria from the user
  - A *list page*, which displays multiple rows, based on the user’s criteria
  - A *create detail page*, which lets the user display or insert one row
  - A *detail page*, which lets the user display, update, or delete one row

The user first encounters the selection criteria page; but if you fail to specify the information needed for that page, the user first encounters the list page, which provides access (in this situation) to every row in the table.

When working in the wizard, you can do as follows:

- Customize the Web pages described earlier, as by varying the fields displayed or including links from one page to another.
- Specify the SQL-record key fields that are used to create, read, update, or delete a row from a given database table or view.
- Customize explicit SQL statements for creating, reading, or updating a row. (The SQL statement for deleting a row cannot be customized.)
- Specify the SQL-record key fields that are used to select a set of rows from a given database or view.
- Customize an explicit SQL statement for selecting a set of rows.
- Validate and run each SQL statement

The output includes these files:

- An HTML file (index.html) that invokes the Web application.
- A set of JSP files that provide the Web pages described earlier.
- An EGL source file that contains all the dataItem parts referenced by the structure items in the SQL record parts.
- For each SQL record part, the wizard also produces two files: one for the record part itself, one for the related, library-based functions. You can reduce the number of files if you select the **Record and library in the same file** check box.

You can customize the Web-based utility after the wizard creates it.

#### Related concepts

"Java program, PageHandler, and library" on page 414

"SQL support" on page 277

#### Related tasks

"Creating a single-table EGL Web application"

"Creating, editing, or deleting a database connection for the EGL wizards" on page 303

"Customizing SQL statements in the EGL wizards" on page 304

"Defining Web pages in the EGL Data Parts and Pages wizard" on page 219

## Creating a single-table EGL Web application

To create an EGL Web application from a single relational database table, do as follows:

1. Select **File > New > Other...** A dialog is displayed for selecting a wizard.
  2. Expand **EGL** and double-click **EGL Data Parts and Pages**. The EGL Data Parts and Pages dialog is displayed.
  3. Enter an EGL Web project name, or select an existing project from the drop-down list. The EGL parts will be generated into this project.
  4. Select an existing database connection from the drop-down list or establish a new database connection--
    - To establish a new database connection, click **Add** and follow the directions in the help topic *Database connection page*, which you can access by pressing F1
    - For details on editing or deleting a database connection, see *Creating, editing, or deleting a database connection for the EGL wizards*
- When a connection is made to the database, a list of database tables is displayed.
5. If you do not want to accept the default EGL file name for data items, type a new file name.
  6. In the **Select your data** field, click on the name of the database table of interest.
  7. In the **Record name** field, either specify the name of the EGL record to be created or accept the default name.
  8. If you want to include the library part and SQL record parts in the same file, select the check box.
  9. To set additional fields to non-default values, click **Next**; otherwise, click **Finish**. The remaining steps assume that you clicked **Next**.
  10. Select the key field to use when reading, updating, and deleting individual rows, then click the right arrow. To select multiple key fields, hold down the

- Ctrl** key while clicking on different field names. To remove a key field from the list on the right, highlight the field name and click the left arrow.
11. Choose the selection condition field to use when selecting a set of rows, then click the right arrow. To select multiple fields, hold down the **Ctrl** key while clicking on different field names. To remove a field from the list on the right, highlight the field name and click the left arrow.
  12. To customize the implicit SQL statements, see *Customizing SQL statements in the EGL wizards*. This option is not available for the EGL **delete** statement.
  13. Click **Next**.
  14. If you want to apply a template to the new Web pages, follow these steps:
    - a. Select the **Select page template** check box.
    - b. Select a page template type by clicking either **Sample page template** or **User-defined page template**.
    - c. Click on the page template you want to use. You can either select a thumbnail or browse to the location of the template by clicking the **Browse** button.
  15. Click **Next**.
  16. To customize the Web pages, see *Defining Web pages in the EGL Data Parts and Pages wizard*.
  17. Click **Next**.
  18. The Generate the Web application screen is displayed, including (at the bottom) a list of the files and Web pages that will be produced:
    - a. To change the name of the EGL Web project that will receive the EGL parts, type a project name in the **EGL Web project name** field or select a project from the related drop-down list.
    - b. To specify the EGL and Java packages for a specific type of part (PageHandler, data, or library), type a package name in the related field or select a name from the related drop-down list.
    - c. To change the name of the JSP and EGL files that are produced for a given Web page, click on the appropriate entry under **Web pages** and type the new name. Each file name includes any letters or numbers that you type, but excludes spaces and other characters.Type or select packages for the PageHandler parts, data parts, and library parts.
  19. Click **Finish**.

#### Related concepts

"SQL support" on page 277

#### Related tasks

"Creating, editing, or deleting a database connection for the EGL wizards" on page 303

"Creating EGL data parts from relational database tables" on page 302

"Customizing SQL statements in the EGL wizards" on page 304

"Defining Web pages in the EGL Data Parts and Pages wizard"

## Defining Web pages in the EGL Data Parts and Pages wizard

The EGL Data Parts and Pages wizard creates a Web application from a relational-database table. When you are working in this wizard, you can specify the following aspects of each type of Web page that is produced:

- Page title
- Style sheet

- Fields to display, including their order and properties
- Links to other pages

When you are working in the wizard dialog called *Define the Web pages of the application*, you can click the tabs to navigate between pages. Do as follows for each page (where possible):

1. Set the page title in the **Page title** field
2. Select a style sheet from a drop-down list in the **Style sheet** field
3. To see the effect of accepting the current page definition, click **Preview**.
4. If you wish to specify the number of rows to display on a page, select **Pagination** and assign the number of rows (a positive integer) to **Page size**.
5. Select the fields to be included on the page:
  - a. To include a field, select the related check box. To select every field, click **All**.
  - b. To exclude a field, clear the related check box. To exclude every field, click **None**.
  - c. To change the display location of a field, click on the field, then use the Up and Down arrows to move the field to another location.
  - d. To set properties for a field, click on the field then double-click the **Value** field in the Properties pane. Enter the value or, in some cases, select from a drop-down list.
6. Select the actions that the user can perform on the page:
  - a. To include an action, select the related check box. To select every action, click **Select All**.

The individual actions are **create**, **delete**, **extract**, **list**, and **read**:

- **Create** links to the create detail page, where the user can display or insert one row. The option is present on the create detail page only to indicate that the user can create a row from that page.
  - **Delete** is available only on the detail page. This option deletes the record that has the key specified by the user.
  - **Extract** links to the selection condition page, which accepts selection criteria from the user. The option is present on the selection condition page only to indicate that the user can cause the return of a result set from that page.
  - **List** links to the list page, which displays multiple rows in accordance with the user's criteria.
  - **Read** is available only on the create detail page. This option displays the record that has the key specified by the user.
  - **Update** is available only on the detail page. This option updates the record that was modified by the user.
- b. To exclude an action, clear the related check box. To exclude every action, click **None**.  
If a given action is always available, you cannot clear the check box.
  - c. To set the Web-page label for an action, click on the action name, then double-click the **Value** field in the Properties pane and enter a value.

#### Related concepts

"SQL support" on page 277

"EGL Data Parts and Pages wizard" on page 217



### Related tasks

- “Creating a single-table EGL Web application” on page 218
- “Creating EGL data parts from relational database tables” on page 302
- “Creating, editing, or deleting a database connection for the EGL wizards” on page 303
- “Setting EGL preferences” on page 115
- “Starting a Web application on the local machine” on page 424

---

## Creating an EGL pageHandler part

A pageHandler part controls a user’s runtime interaction with a Web page by providing data and services to a Java Server Faces JSP. When you create a JSP, a pageHandler part is automatically created for you in a package called *pagehandlers* within the *EGLSource* folder. The name of the pageHandler part is the same as the corresponding JSP, but with a .egl file extension.

Optionally, you can create a pageHandler part and let the system automatically add the JSP to your project, provided a JSP file with the same name does not already exist within the EGL Web project. To create an EGL pageHandler part, do as follows:

1. If your EGL Web project does not contain a package named *pagehandlers*, you must create one. Page Designer requires that all pageHandler parts reside in a package named *pagehandlers*. For details on creating packages, see *Creating an EGL package*.
2. Identify an EGL file within the *pagehandlers* package to contain the pageHandler part. Open the file in the EGL editor. You must create an EGL file if you do not already have one.
3. Type the specifics of the pageHandler part according to EGL syntax (for details, see *PageHandler part in EGL source format*). You can use content assist to place an outline of the pageHandler part syntax in the file.
4. Save the EGL file.

### Related concepts

- “EGL projects, packages, and files” on page 15
- “PageHandler” on page 223

### Related tasks

- “Creating an EGL package” on page 130
- “Creating an EGL source file” on page 130
- “Using the EGL templates with content assist” on page 131
- “Using the Quick Edit view for PageHandler code” on page 244

### Related reference

- “Content assist in EGL” on page 577
- “Naming conventions” on page 778
- “PageHandler part in EGL source format” on page 785

## Page Designer support for EGL

When you create a JSP file in an EGL Web project, EGL automatically creates a PageHandler, and that PageHandler includes skeletal EGL code for you to customize. Then, in Page Designer, do as follows:

1. Drag components from the palette to a JSP
2. Use the Attributes view to set component-specific characteristics such as color and to set up *bindings*, which are relationships between components and either data or logic



You can do work that is specific to EGL:

- Create EGL variables and place them in an existing PageHandler.
- Bind PageHandler items to JSP user-interface components.
- Bind PageHandler functions to buttons and hyperlink controls. The functions act as event handlers.

When you use the source tab in Page Designer, you can manually bind components in a JSP file (specifically, in a JavaServer Faces file) to data areas and functions in a PageHandler. Although EGL is not case sensitive, EGL names referenced in the JSP file must have the same case as the EGL variable or function declaration; and if you fail to maintain an exact match, a JavaServer Faces error occurs. It is recommended that you avoid changing the case of an EGL variable or function after you bind that variable or function to a JSP field.

For further details on naming issues, see *Changes to EGL identifiers in JSP files and generated Java beans*.

### Binding components to data areas in the PageHandler

Most components on the JSP have a one-to-one correspondence with data. A text box, for example, shows the content of the EGL item to which the text box is bound. An input text box also updates the EGL item if the user changes the data.

A more complex situation occurs when you are specifying a check box group, list box, radio button group, or combo box. In those cases, you need two different kinds of bindings:

- One is to bind the component to the text that you want to display to the user. An example is the text of an item in a list box.
- One is to bind the component to a PageHandler data area that receives a value to indicate the user's choice. You might create a data item, for example, to receive the numeric index of a user-selected list-box item.

In the Properties view, you can follow either of two procedures to bind the component to the text that the user sees:

- You can use **Add Choice** to indicate that the component is associated with a single character string, which may be specified explicitly or by identifying a PageHandler item
- You can use **Add Set of Choices** to indicate that the component is associated with a list of character strings, which may be specified explicitly or by identifying a PageHandler area such as a data table or an array of character items

Alternatively, you can bind a single-select component (combo box, single-select list box, or radio button group) to an array of character items by dragging the array from the Page Data view to the component.

To bind a component to a data area that will receive a value indicating the user's choice, you can work in either the Page Data view or the Properties view. The procedure is the same as when you are binding any component, even a simple text box.

If the value can be only one of two alternatives, you can bind the component to an EGL item for which the item property **boolean** is set to *yes*. The component populates the item with one of two values:

- For a character item, the value is **Y** (for yes) or **N** (for no)

- For a numeric item, the value is **1** (for yes) or **0** (for no)

When a check box is displayed, the status (whether checked or not) is dependent on the value in the bound item.

For details on the properties that can be applied to data items in the PageHandler, see *PageHandler field properties*.

## Binding components to functions

After you drag a command button or a command hyperlink to the page surface, you can bind that component to an existing EGL function or to an event handler that the Page Designer creates:

- You can bind the component to an existing event handler in any of these ways--
  - By dragging the EGL function from the Actions node in the Page Data view to the component, as is recommended
  - By opening the component in the Quick Edit view
  - By right-clicking on the component and selecting **Edit Faces Command Event**
- You can cause Page Designer to create a new event handler either when you open the component in the Quick Edit view or when you right-click on the component and select **Edit Faces Command Event**

If the Page Designer creates an event handler in the PageHandler and gives you access to that PageHandler function, the name of the function is the tool-assigned button ID plus the string "Action". If the name is not unique to the PageHandler, the Page Designer appends a number to the function name.

### Related concepts

"PageHandler"

### Related tasks

"Creating an EGL field and associating it with a Faces JSP" on page 242

"Associating an EGL record with a Faces JSP" on page 243

"Using the Quick Edit view for PageHandler code" on page 244

### Related reference

"PageHandler part in EGL source format" on page 785

"PageHandler field properties" on page 792

## PageHandler

An EGL *PageHandler* is an example of *page code*; it controls a user's runtime interaction with a Web page and can do any of these tasks:

- Assign data values for submission to a JSP file. Those values are ultimately displayed on a Web page.
- Change the data returned from the user or from a called program.
- Forward control to another JSP file.

You can work most easily by customizing a JSP file and creating the PageHandler in Page Designer; for details, see *Page Designer support for EGL*.

The PageHandler itself includes variables and the following kinds of logic--

- An OnPageLoad function, which is invoked the first time that the JSP renders the Web page

- A set of event handler functions, each of which is invoked in response to a specific user action (specifically, by the user clicking a button or hypertext link)
- Optionally, validation functions that are used to validate Web-page input fields
- Private functions that can be invoked only by other PageHandler functions

**Note:** The OnPageLoad function is identified in PageHandler property onPageLoadFunction; the name can be any valid EGL function name. The OnPageLoad function automatically retrieves any user-supplied arguments that were passed to it; can invoke other functions or call other programs; and can place additional data in the request or session object of the Web application server; but the function can neither forward control to another page nor cause an error message to be displayed when the page is first presented to the user.

The variables in the PageHandler are accessed in two ways:

- The runtime environment accesses the data automatically. If a field in the JSP is *bound* to a field in the PageHandler, the result is as follows--
  - After the OnPageLoad function runs and before the Web page is displayed, each PageHandler field value is written to the JSP field to which the data is bound.
  - When the user submits a form in which bound JSP fields reside, the value in each field of the submitted form is copied to the associated PageHandler field. Only then is control passed to an event handler. (However, this description does not include the validation steps, which are covered later in this topic.)
- The event handlers and the OnPageLoad function also can interact with the data, as well as with data stores (such as SQL databases) and with called programs.

The PageHandler part should be simple. Although the part might include lightweight data validations such as range checks, you are advised to invoke other programs to perform complex business logic. Database access, for example, should be reserved to a called program.

### Output associated with a PageHandler

When you save a PageHandler, EGL places a JSP file in the project folder WebContent\WEB-INF, but only in this case:

- You assigned a value to the PageHandler **view** property, which specifies a JSP file name
- The folder WebContent\WEB-INF does not contain a JSP file of the specified name

When generating a PageHandler, EGL never overwrites a JSP file.

If a Workbench preference is set to automatic build on save, PageHandler generation occurs whenever you save the PageHandler. In any case, when you generate a PageHandler, the output is composed of the following objects:

- The *page bean* is a Java class that contains data and that provides initialization, data validation, and event-handling services for the Web page. In the documentation that refers to runtime events, the word *PageHandler* is sometimes used to refer to the page bean.
- A <managed-bean> element is placed in the JSF configuration file in your project, to identify the page bean at run time.

- A <navigation-rule> element is created in the JSF application configuration file to associate a JSF outcome (the name of the PageHandler) with the JSP file to be invoked.
- A JSP file, in the same situation as when you save the PageHandler.

All data tables and records that are used by the part handler are also generated.

## Validation

If the JSP-based JSF tags perform data conversion, validation, or event handling, the JSF runtime does the necessary processing as soon as the user submits the Web page. If errors are found, the JSF runtime may re-display the page without passing control to the PageHandler. If the PageHandler receives control, however, it may conduct a set of EGL-based validations.

The EGL-based validations occur if you specify the following details when you declare the PageHandler:

- The element edits (such as minimum input length) for individual input fields.
- The type-based edits (character, numeric) for individual fields.
- The DataTable edits (range, match valid, and match invalid) for individual input fields, as explained in *DataTable part*.
- The validator functions for individual input fields.
- The validator function for the PageHandler as a whole.

The PageHandler oversees the edits in the following order, but only for fields whose values were changed by the user:

1. All the elementary and type-based edits, even if some fail
2. (If the prior edits were successful) all the table edits, even if some fail
3. (If the prior edits were successful) all the field-edit functions, even if some fail
4. (If all prior edits were successful) the PageHandler edit function

The PageHandler field property **validationOrder** defines the order in which both the individual input fields are edited and the field validator functions are invoked.

If no validationOrder properties are specified, the default is the order of fields defined in the PageHandler, from top to bottom. If validationOrder is defined for some but not all of the fields in a PageHandler, validation of all fields with the validationOrder property occurs first, in the specified order. Then, validation of fields without the validationOrder property occurs in the order of fields in the PageHandler, from top to bottom.

If the EGL runtime finds an error outside of a validator function, the JSF runtime code re-displays the same Web page with embedded error messages. If a validator function finds an error, the function can forward control to another Web page, but the default behavior is to re-display the same Web page.

## Runtime scenario

This section gives a technical overview of the runtime interaction of user and Web application server.

When the user invokes a JSP that is supported by a PageHandler, the following steps occur:

1. The Web application server initializes the environment--

- a. Constructs a session object to retain data that is needed across multiple interactions with the user:
  - If the PageHandler **scope** property is set to *session*, the page bean will be assigned to the session object.
  - In any case, EGL provides the following system functions, which let you place data into the session object and let you retrieve or clear data from the session object: **J2EELib.setSessionAttr**, **J2EELib.getSessionAttr**, and **J2EELib.clearSessionAttr**.
- b. Constructs a request object to retain data on the user's current interaction with the user:
  - If the PageHandler **scope** property is set to *request*, the page bean will be assigned to the request object
  - In any case, EGL provides the following system functions, which let you place data into the request object and let you retrieve or clear data from the request object: **J2EELib.setRequestAttr**, **J2EELib.RequestAttr**, and **J2EELib.clearRequestAttr**
- c. Initializes the JSF runtime code

**Note:** Even when a PageHandler **scope** property is set to *session*, the data in that PageHandler is not necessarily available throughout the user session. If PageHandler A forwards control to PageHandler B, for example, the data in PageHandler A is no longer available unless you took action like the following:

- Included the data in arguments that are passed to the new PageHandler; or
  - Included the data in the session or request object; or
  - Included the data in a relational database or other data store.
2. The JSF runtime code acts as follows--
    - a. Invokes the JSP file, which a JSF configuration table associates with a particular page bean
    - b. Creates the page bean
    - c. Assigns the page bean to the session or request object, depending on the value of the PageHandler **scope** property
    - d. Gives control to the page bean's onPageLoad function (if any), providing any user-specified arguments
  3. The JSP accesses data directly from the page bean, for inclusion in the Web page; then displays the Web page to the user, leaving the session object in place and (more temporarily) leaving the request object in place
  4. The user might (for example) supply data in the on-screen fields associated with an HTML <FORM> tag, then click the SUBMIT button to invoke a PageHandler function.
  5. The JSF runtime acts as follows:
    - a. Destroys the request object; and, if the **scope** property for the PageHandler was set to *request*, removes the page bean
    - b. Constructs a new request object
    - c. Does the next steps only if the PageHandler scope was set to *request*:
      - 1) Creates the page bean
      - 2) Assigns the page bean to the request object
      - 3) Gives control to the page bean's onPageLoad function (if any), providing any user-specified arguments

- d. Handles JSF validation (but only a subset of JSF validation is possible in the context of EGL PageHandler processing):
  - Places the user data into the page bean (if the input was found to be valid from a JSF point of view); or
  - Re-displays the Web page with JSF messages (if the input was found to be invalid). The subsequent steps in this scenario occur if the input data was valid.
- e. Places the data received from the submitted form into the page bean, for EGL validation (which is described earlier). If no error occurs, the JSF runtime code invokes the PageHandler function requested by the user:
  - If the function ends without issuing a **forward** statement, the JSF runtime code re-displays the same Web page without re-invoking the OnPageLoad function.
  - If the function ends with a **forward** statement that refers to the same Web page, the situation is the same as in the previous case; *the OnPageLoad function is not re-invoked*
  - If the function ends with a **forward** statement that refers to a different page, the data in the page bean is lost, although you may pass arguments and retain values in either the session or request object

#### Related concepts

“References to parts” on page 23

“Web support” on page 215

#### Related reference

“Page Designer support for EGL” on page 221

“PageHandler field properties” on page 792

“PageHandler part in EGL source format” on page 785

“PageHandler part properties” on page 788

## Supporting multiple languages for labels or help text in a PageHandler

When you are developing a JSP file for use with a PageHandler, you can specify literal text for labels, as well as for the hover-help text that is displayed when the user places the cursor over an input field. When you are developing the PageHandler itself, you can set the default values for that literal text by setting the primitive field-level properties **displayName** and **help** for a given field.

When you customize the JSP file, however, you can identify a Java resource bundle or properties file that provides text at run time. The process is as follows:

1. Create the resource bundle or properties file.
2. Add a JSF loadBundle tag to the Web page:
 

```
<f:loadBundle baseName=fileName var=variableName>
```

*fileName*

Name of the resource bundle or properties file, not including the locale

*variableName*

Name of the variable to be used in other JSF tags in the Web page.

An example is as follows:

```
<f:loadBundle baseName="Resources" var="labels"/>
```

3. To access the resource bundle, reference the variable in an output field, whether for a label or help text::

```
<h:output_text value="#{variableName.key}"
```

*variableName*  
Name of a variable, as specified in a loadBundle tag.

*key*  
The key for the message in the resource bundle.

An example is as follows:

```
<h:output_text value="#{labels.label1}"
```

To identify the resource bundle or properties file used for runtime messages, set the PageHandler part property **msgResource**.

#### Related concepts

“PageHandler” on page 223

#### Related reference

“displayName” on page 807

“help” on page 810

“PageHandler part properties” on page 788

## JavaServer Faces controls and EGL

JavaServer Faces (JSF) is a server-side user interface component framework. In simple terms, JSF is a set of tools and components that allow you to create interfaces for Web pages. JSF components can display data on a Web page and accept input from the user.

This topic explains the relationship between JSF components and EGL. For more details on JSF, see *Creating Faces applications - overview*. To see a related tutorial, click **Help > Tutorials Gallery**; expand **Do and Learn**; and select *Display dynamic information on Web pages with JavaServer Faces*.

Two methods are available for displaying EGL data on a Web page using JSF controls:

- You can create JSF controls automatically from data items in the Page Data view or from data items you create in the Page Designer view. To use this method, select the check box named **Add controls to display the EGL element on the Web page** as you follow the directions in *Associating an EGL record with a Faces JSP* or *Creating an EGL data item and associating it with a Faces JSP*.
- You can add JSF controls manually and bind them to data in the Page Data view. This method allows you to customize the layout of the JSF controls on the page, rather than using the default layout. To use this method, see one of the following topics:
  - *Binding a JavaServer Faces input or output component to an EGL PageHandler*
  - *Binding a JavaServer Faces check box component to an EGL PageHandler*
  - *Binding a JavaServer Faces single-selection component to an EGL PageHandler*
  - *Binding a JavaServer Faces multiple-selection component to an EGL PageHandler*

You can also bind EGL functions in PageHandlers to JSF controls. See *Binding a JavaServer Faces command component to an EGL PageHandler*.

#### Related tasks

“Creating an EGL field and associating it with a Faces JSP” on page 242

“Associating an EGL record with a Faces JSP” on page 243

“Binding a JavaServer Faces command component to an EGL PageHandler” on page 244

“Binding a JavaServer Faces input or output component to an EGL PageHandler” on page 245



“Binding a JavaServer Faces check box component to an EGL PageHandler” on page 246  
“Binding a JavaServer Faces single-selection component to an EGL PageHandler” on page 247  
“Binding a JavaServer Faces multiple-selection component to an EGL PageHandler” on page 248

#### Related reference

“Page Designer support for EGL” on page 221

### JSF component tree

EGL allows you to dynamically update aspects of JSF controls that are displayed in a Web browser. For example, you can change the color of a text box if the user enters invalid information in that text box. These changes occur on the Web application server, affecting the information available to the JSP that in turn presents the Web page to the browser.

EGL accesses JSF components through the page’s *JSF component tree*, an XML-based structure that identifies the JSF components specified in the JSP file. When using the JSF component tree, you associate an EGL variable with the JSF component you want to access. Then, you can perform the same functions on that EGL variable that you can perform on the JSF component itself, such as changing the style and other properties. For a complete list of the JSF components and the functions that are available for each of those components, see the documentation for Java Server Faces at [http://java.sun.com/j2ee/javaserverfaces/1.1\\_01/docs/api/index.html](http://java.sun.com/j2ee/javaserverfaces/1.1_01/docs/api/index.html).

In general, you can access the JSF controls on a Web page by following these steps. For more information and a complete example, see *Accessing a JSF component from a pageHandler*.

1. Create a EGL Web project with JSF component interface support or add JSF component interface support to an existing EGL Web project. See *Adding JSF component interface support to an EGL Web project*.
2. Create a Faces JSP file and add one or more JSF components to it.
3. In the page’s page code file, add the following code. If you created the Faces JSP file after you added support for the JSF component interface to the project, this code is added to the page code file automatically.
  - Add the following import statement:  

```
import com.ibm.egl.jsf.*
```
  - Within the page’s pageHandler, create a variable of type `UIViewRoot`.
  - Specify the name of the `UIViewRoot` variable in the pageHandler property **viewRootVar**.
4. Create a variable of the JSF control type you want to access or have the source assistant create one for you. For more information, see *Accessing a JSF component with the source assistant*. For example, a variable of type `HtmlInputText` refers to a JSF input field on the page. The following code creates a variable of type `HtmlInputText`:  

```
myControl HtmlInputText;
```
5. Link the variable to the JSF component, using the `UIViewRoot` variable. For example, the following code links a variable of type `HtmlInputText` to a JSF text input field named `inputField1` inside a form named `form1`:  

```
myControl = myViewRoot.findComponent("form1:inputField1");
```
6. Use the variable to change the JSF component. For example, the following code changes the text in an input field to the color red:  

```
myControl.setStyle("color : red");
```



### Related concepts

“viewRootVar property”

### Related tasks

“Adding JSF component interface support to an EGL Web project”

“Accessing a JSF component from a pageHandler” on page 232

“Changing the style class of a JSF component” on page 235

“Changing the style of a JSF component” on page 234

“Changing the target of a JSF link” on page 236

“Enabling or disabling JSF components” on page 237

“Setting the size of a JSF image” on page 238

“Setting event handlers for a JSF component” on page 239

“Setting JSF data table properties” on page 241

### Related reference

“Component tree access parts” on page 231

**Adding JSF component interface support to an EGL Web project:** Before you can access JSF components on a Faces JSP file, your EGL Web project must contain the packages that allow you to access the JSF component tree. There are two ways to add these packages to your Web project:

- Create a new EGL Web project and select the **EGL support with JSF** and the **EGL support with JSF Component Interfaces** check boxes on the Features page of the New EGL Web Project wizard.
- Add JSF component interface support to an existing EGL Web project.

To add support for the JSF component interface to an existing EGL Web project, follow these steps:

1. In the Project Explorer view, right-click the EGL Web project and then click **Properties**. The Properties window opens.
2. In the Properties window, click **Web Project Features**.
3. Select the **EGL support with JSF Component Interfaces** check box.
4. Click **OK**.

You can not remove support for the JSF component interface from a project.

### Related concepts

“JSF component tree” on page 229

### Related tasks

“Creating an EGL project” on page 127

“Accessing a JSF component from a pageHandler” on page 232

### Related reference

“Component tree access parts” on page 231

**viewRootVar property:** In a pageHandler part, the **viewRootVar** property indicates a variable of the type `UIViewRoot` to represent the root of the page’s JSF component tree. Once the **viewRootVar** variable is defined, you can define other EGL variables to represent JSF components on the page, using the **viewRootVar** variable to link the variables to the JSF components. For more information, see *Accessing a JSF component from a pageHandler*.

The **viewRootVar** property is required for pageHandlers that access the JSF component tree. Also, if the **viewRootVar** property is specified, the pageHandler must create a variable of the type `UIViewRoot` with the name specified in the **viewRootVar** property.

To assign an EGL variable to a JSF component, use the `findComponent` function on the `viewRootVar` variable. For example, the following code creates an EGL variable and links it to a text input field using the `viewRootVar` variable:

```
myInputField HtmlInputText;  
myInputField = myViewRoot.findComponent("form1:text1");
```

#### Related concepts

"JSF component tree" on page 229

"Overview of EGL properties" on page 64

#### Related tasks

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a pageHandler" on page 232

#### Related reference

"Component tree access parts"

**Component tree access parts:** To access a JSF component on a Faces JSP page, you first define an EGL variable of that component's type. Then, you assign that variable to the JSF component. The source assistant can create this code for you most of the time. For more information, see *Accessing a JSF component from a pageHandler*.

Use the following syntax to assign a variable to a JSF control:

```
controlVar = findComponent(controlName);
```

*controlVar*

The variable of the appropriate type

*controlName*

A string variable or literal that identifies the control. In most cases, you specify a *series* of JSF control IDs, starting with the ID of the top-level form in the component tree and continuing to the ID of the specific control, with each ID separated from the next by a colon (:). The example used the following string to reference a text box named *text1* in a top-level form named *form1*:

```
"form1:text1"
```

Once you have assigned the variable to the JSF control, you can use functions on that variable to make changes to the JSF component. Different types of JSF controls accept different functions.

#### Related concepts

"JSF component tree" on page 229

"viewRootVar property" on page 230

#### Related tasks

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a pageHandler" on page 232

"Changing the style class of a JSF component" on page 235

"Changing the style of a JSF component" on page 234

"Changing the target of a JSF link" on page 236

"Enabling or disabling JSF components" on page 237

“Setting the size of a JSF image” on page 238  
“Setting event handlers for a JSF component” on page 239

## Accessing a JSF component from a pageHandler

You can use EGL code to call Java functions recognized by JSF components. In this way, you can change the appearance and behavior of these components from an EGL pageHandler. The following is an example of a page code file that includes EGL code to access a JSF component:

```
package pagehandlers;

import com.ibm.egl.jsf.*;

pageHandler myPageHandler
{onPageLoadFunction = onPageLoad,
 view = "myPage.jsp",
 viewRootVar = "myViewRoot"}

myViewRoot UIViewRoot;

function onPageLoad()
myInputField HtmlInputText;
myInputField = myViewRoot.findComponent("form1:text1");
myInputField.setStyle("color : red");
end

end
```

Follow these steps to access a JSF component from a pageHandler:

1. Make sure that your EGL Web project has support for the JSF component interface. See *Adding JSF component interface support to an EGL Web project*.
2. Create a Faces JSP file and add one or more JSF components to it.
3. Optionally, you may want to change the ID attribute of the JSF components so they will be easier to find from the EGL code. You can change the ID attribute by selecting the component and entering a meaningful mnemonic in the **ID** field in the Properties view.
4. In the page's page code file, add the following code. If you created the Faces JSP file after you added support for the JSF component interface to the project, this code is added to the page code file automatically.

- Add the following import statement:

```
import com.ibm.egl.jsf.*
```

The packages imported by this statement contain a series of interface parts, each of which provides access to Java code. You do not need to understand the interface mechanism, though an explanation is provided in *EGL interface technology*.

- Within the page's pageHandler, declare a variable of type UIViewRoot.
  - Specify the name of the UIViewRoot variable in the pageHandler property **viewRootVar**.
5. On a blank line inside a function in the pageHandler, press **Ctrl+Shift+Z**. The EGL Source Assistant window opens, displaying the JSF components on the page.
  6. In the EGL Source Assistant window, select the JSF component you want to access.
  7. Click **OK**.

The EGL source assistant adds two lines of EGL code to the pageHandler:

- The first line of code defines an EGL variable of the type that matches the JSF component that you selected. In the above example, a variable of type `HtmlInputText` is defined to access a JSF input text field, using this code:  

```
myInputField HtmlInputText;
```
  - The second line of code associates that variable with the JSF component. In the above example, the variable is associated with a JSF input text field named `text1` that is located within a form named `form1`, using this code:  

```
myInputField = myViewRoot.findComponent("form1:text1");
```
8. Use the variable to change the JSF component. For example, the following code uses the `setStyle` function to change the text in an input field to the color red:  

```
myInputField.setStyle("color : red");
```

  
When this code runs, the style of the input field is changed. In this example, the HTML code displayed by the browser looks like this:  

```
<input id="form1:text1" type="text" name="form1:text1" style="color : red" />
```

Following are some notes about accessing JSF components with EGL code:

- For the complete list of JSF functions accessible in EGL, open the file `FacesHtmlComponent.egl` in the package `com.ibm.egl.jsf`. This file is added to your project when you add support for the JSF component interface. The functions are briefly explained in comments to this file. For more detailed information, see the documentation for Faces components.
- When passing a parameter to one of these functions, be sure to pass the correct data type. Because many of the parameters passed to these functions are inserted into HTML attributes, they must be passed EGL string variables, even if the name of the function suggests that the parameter is a numerical or boolean value.

For example, the `setWidth` function sets the width of a component in pixels, or in a percentage of its original size if the percent (%) symbol is appended. Because this function's parameter is a measurement, it might seem to take an integer data type as a parameter. However, this function must receive a string. To set the width of a component to 300 pixels, you must pass a string variable with the value "300".

- Because many of the functions set or return information from HTML attributes, you should be aware of the HTML attributes connected to the functions you are using. You may change an HTML attribute that is needed for the page to work properly. For example, if you change the style class of a component as in *Changing the style class of a JSF component*, that new style class of the component must be available to the page in a CSS file or style tag, or else the component may not display properly. Be sure to test any changes you make to Web pages. The comments in the `FacesHtmlComponent.egl` file note functions that change HTML attributes directly.
- In most cases, the changes you make to the JSF components are not cumulative. For example, if you set a component's text to red with the code `myComponent.setStyle("color: red");` and then set the same component's text to bold with the code `myComponent.setStyle("font-weight: bold");`, the text will be bold but not red, because the change to bold overwrites the change to red.

To add several changes to a JSF component, retrieve the current state of the component and append the new data, paying attention to how the list of changes is delimited. For example, use the following code to change a component's text to bold and red, without overwriting any previous changes to that component's style:

```
myStyleString string;
myStyleString = myComponent.getStyle() + "; color: red; font-weight: bold";
myComponent.setStyle(myStyleString);
```

There are many different changes you can make to JSF components. See the related tasks for some examples.

### Related concepts

“JSF component tree” on page 229

“viewRootVar property” on page 230

### Related tasks

“Adding JSF component interface support to an EGL Web project” on page 230

“Changing the style class of a JSF component” on page 235

“Changing the style of a JSF component”

“Changing the target of a JSF link” on page 236

“Enabling or disabling JSF components” on page 237

“Setting the size of a JSF image” on page 238

“Setting event handlers for a JSF component” on page 239

“Setting JSF data table properties” on page 241

### Related reference

“Component tree access parts” on page 231

**Changing the style of a JSF component:** You can change the appearance of a JSF component with EGL code, such as changing the text color. To make a larger change in the component’s appearance by changing its style class, see *Changing the style class of a JSF component*.

This task has the following prerequisites. For more information, see *Accessing a JSF component from a pageHandler*.

- Your EGL Web project must have support for the JSF component interface. See *Adding JSF component interface support to an EGL Web project*.
- The Faces JSP’s page code file must have the following import statement:  
`import com.ibm.egl.jsf.*`
- You must declare a variable of type `UIViewRoot` within the pageHandler.
- You must specify the name of the `UIViewRoot` variable in the pageHandler property **viewRootVar**.

Follow these steps to change the style of a JSF component from an EGL pageHandler:

1. On a blank line inside a function in the pageHandler, press **Ctrl+Shift+Z**. The EGL Source Assistant window opens, displaying the JSF components on the page.
2. In the EGL Source Assistant window, select the JSF component you want to access.
3. Click **OK**.

The EGL source assistant adds two lines of EGL code to the pageHandler. The first line defines an EGL variable of the type that matches the JSF component that you selected. The second line associates that variable with the JSF component. For example, the code to access a JSF input text component might look like this:

```
text1 HtmlInputText;
text1 = myViewRoot.findComponent("form1:text1");
```

4. Using the EGL variable created by the source assistant, change the style of the JSF component with the `setStyle` function. For example, to change a text field's text to red, add this code:

```
text1.setStyle("color : red");
```

When this code runs, the style attribute of the input field is changed. In this example, the HTML code displayed by the browser looks like this:

```
<input id="form1:text1" type="text" name="form1:text1" style="color : red" />
```

The new style attribute overwrites any previous style attribute. To make more than one change to the component's style, separate changes with semicolons (;). For example, to change the color to red and the size to 20 points, use this code:

```
text1.setStyle("color : red; font-size: 20pt");
```

Following are some examples of other changes you can make to the style of a component. Not all styles are compatible with all JSF components.

**`text1.setStyle("font-size : 20pt");`**

Sets the size of the font in the component to 20 points.

**`text1.setStyle("text-align: center");`**

Centers the text within the component.

**`text1.setStyle("border-style : solid; border-color : red");`**

Adds a red border composed of a solid line around the component.

**`text1.setStyle("font-weight : bold");`**

Makes the text within the component bold.

**`text1.setStyle("height : 50px");`**

Makes the component 50 pixels tall.

#### **Related concepts**

"JSF component tree" on page 229

"viewRootVar property" on page 230

#### **Related tasks**

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a pageHandler" on page 232

"Changing the style class of a JSF component"

"Changing the target of a JSF link" on page 236

"Enabling or disabling JSF components" on page 237

"Setting the size of a JSF image" on page 238

"Setting event handlers for a JSF component" on page 239

"Setting JSF data table properties" on page 241

#### **Related reference**

"Component tree access parts" on page 231

**Changing the style class of a JSF component:** Like many elements on a Web page, JSF components can be assigned a style class with the `class` attribute. A style class, not to be confused with a Java class, is a group of zero to many commands that describes the appearance of an element on a Web page. Style classes are defined with the Cascading Style Sheets language, a language that can control many different aspects of the appearance of a Web page.

You can change the style class of a JSF component on a Faces JSP file from one class to another. To make smaller changes to a JSF component's style, such as changing the text color, see *Changing the style of a JSF component*.

This task has the following prerequisites. For more information, see *Accessing a JSF component from a pageHandler*.

- Your EGL Web project must have support for the JSF component interface. See *Adding JSF component interface support to an EGL Web project*.
- The Faces JSP's page code file must have the following import statement:  

```
import com.ibm.egl.jsf.*
```
- You must declare a variable of type `UIViewRoot` within the pageHandler.
- You must specify the name of the `UIViewRoot` variable in the pageHandler property **viewRootVar**.

Follow these steps to change the style class of a JSF component from an EGL pageHandler:

1. On a blank line inside a function in the pageHandler, press **Ctrl+Shift+Z**. The EGL Source Assistant window opens, displaying the JSF components on the page.
2. In the EGL Source Assistant window, select the JSF component you want to access.
3. Click **OK**.

The EGL source assistant adds two lines of EGL code to the pageHandler. The first line defines an EGL variable of the type that matches the JSF component that you selected. The second line associates that variable with the JSF component. For example, the code to access a JSF input text component might look like this:

```
text1 HtmlInputText;  
text1 = myViewRoot.findComponent("form1:text1");
```

4. Using the EGL variable created by the source assistant, set the style class of the JSF component with the `setStyleClass` function. For example, to set a text field to a style class named `errorField`, add this code:

```
text1.setStyleClass("errorField");
```

When this code runs, the style class of the input field is changed. In this example, the HTML code displayed by the browser looks like this:

```
<input id="form1:text1" type="text" name="form1:text1" class="errorField" />
```

### Related concepts

"JSF component tree" on page 229

"viewRootVar property" on page 230

### Related tasks

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a pageHandler" on page 232

"Changing the style of a JSF component" on page 234

"Changing the target of a JSF link"

"Enabling or disabling JSF components" on page 237

"Setting the size of a JSF image" on page 238

"Setting event handlers for a JSF component" on page 239

"Setting JSF data table properties" on page 241

### Related reference

"Component tree access parts" on page 231

**Changing the target of a JSF link:** You can change the target attribute of a JSF link from a pageHandler. For example, you can set the link's target attribute to `_blank` to make that link open in a new browser window.



This task has the following prerequisites. For more information, see *Accessing a JSF component from a pageHandler*.

- Your EGL Web project must have support for the JSF component interface. See *Adding JSF component interface support to an EGL Web project*.
- The Faces JSP's page code file must have the following import statement:  

```
import com.ibm.egl.jsf.*
```
- You must declare a variable of type `UIViewRoot` within the `pageHandler`.
- You must specify the name of the `UIViewRoot` variable in the `pageHandler` property **viewRootVar**.

Follow these steps to change the target attribute of a JSF link from a `pageHandler`:

1. On a blank line inside a function in the `pageHandler`, press **Ctrl+Shift+Z**. The EGL Source Assistant window opens, displaying the JSF components on the page.
2. In the EGL Source Assistant window, select the JSF component you want to access.
3. Click **OK**.

The EGL source assistant adds two lines of EGL code to the `pageHandler`. The first line defines an EGL variable of the type that matches the JSF link that you selected. The second line associates that variable with the JSF link. For example, the code might look like this:

```
linkEx1 HtmlOutputLink;  
linkEx1 = myViewRoot.findComponent("form1:linkEx1");
```

4. Using the EGL variable created by the source assistant, change the target of the link with the `setTarget()` function. For example, to make the link open in a new window, add this code:

```
linkEx1.setTarget("_blank");
```

### Related concepts

"JSF component tree" on page 229

"viewRootVar property" on page 230

### Related tasks

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a pageHandler" on page 232

"Changing the style class of a JSF component" on page 235

"Changing the style of a JSF component" on page 234

"Enabling or disabling JSF components"

"Setting the size of a JSF image" on page 238

"Setting event handlers for a JSF component" on page 239

"Setting JSF data table properties" on page 241

### Related reference

"Component tree access parts" on page 231

**Enabling or disabling JSF components:** You can enable or disable JSF input fields and command buttons with EGL code. A disabled component can not be edited or changed on the Web page.

This task has the following prerequisites. For more information, see *Accessing a JSF component from a pageHandler*.

- Your EGL Web project must have support for the JSF component interface. See *Adding JSF component interface support to an EGL Web project*.



- The Faces JSP's page code file must have the following import statement:  
`import com.ibm.egl.jsf.*`
- You must declare a variable of type `UIViewRoot` within the `pageHandler`.
- You must specify the name of the `UIViewRoot` variable in the `pageHandler` property **`viewRootVar`**.

Follow these steps to enable or disable a JSF component from an EGL `pageHandler`:

1. On a blank line inside a function in the `pageHandler`, press **Ctrl+Shift+Z**. The EGL Source Assistant window opens, displaying the JSF components on the page.
2. In the EGL Source Assistant window, select the JSF component you want to access.
3. Click **OK**.

The EGL source assistant adds two lines of EGL code to the `pageHandler`. The first line defines an EGL variable of the type that matches the JSF component that you selected. The second line associates that variable with the JSF component. For example, the code to access a JSF input text component might look like this:

```
text1 HtmlInputText;
text1 = myViewRoot.findComponent("form1:text1");
```

4. Using the EGL variable created by the source assistant, enable or disable the JSF component with the `setEnabled` function. For example, to enable a text field, add this code:

```
text1.setEnabled(no);
```

To disable the text field, add this code:

```
text1.setEnabled(yes);
```

### Related concepts

"JSF component tree" on page 229

"viewRootVar property" on page 230

### Related tasks

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a `pageHandler`" on page 232

"Changing the style class of a JSF component" on page 235

"Changing the style of a JSF component" on page 234

"Changing the target of a JSF link" on page 236

"Setting the size of a JSF image"

"Setting event handlers for a JSF component" on page 239

"Setting JSF data table properties" on page 241

### Related reference

"Component tree access parts" on page 231

**Setting the size of a JSF image:** You can change the size of a JSF image on a Faces JSP page with EGL code. You must use a Faces image component; ordinary HTML image tags can not be changed directly from an EGL `pageHandler`.

This task has the following prerequisites. For more information, see *Accessing a JSF component from a `pageHandler`*.

- Your EGL Web project must have support for the JSF component interface. See *Adding JSF component interface support to an EGL Web project*.

- The Faces JSP's page code file must have the following import statement:  
`import com.ibm.egl.jsf.*`
- You must declare a variable of type `UIViewRoot` within the `pageHandler`.
- You must specify the name of the `UIViewRoot` variable in the `pageHandler` property **`viewRootVar`**.

Follow these steps to change the size of a JSF image component from an EGL `pageHandler`:

1. On a blank line inside a function in the `pageHandler`, press **Ctrl+Shift+Z**. The EGL Source Assistant window opens, displaying the JSF components on the page.
2. In the EGL Source Assistant window, select the JSF image component you want to access.
3. Click **OK**.

The EGL source assistant adds two lines of EGL code to the `pageHandler`. The first line defines an EGL variable of the type that matches the JSF component that you selected. The second line associates that variable with the JSF component. For example, the code to access a JSF image component might look like this:

```
imageEx1 HtmlGraphicImageEx;  
imageEx1 = myViewRoot.findComponent("imageEx1");
```

4. Using the EGL variable created by the source assistant, change the size of the JSF image component with the `setHeight` and `setWidth` functions, passing each function a string or literal that specifies the measurement in pixels. For example, to make the image 300 pixels wide and 200 pixels tall, add this code:

```
imageEx1.setWidth("300");  
imageEx1.setHeight("200");
```

### Related concepts

"JSF component tree" on page 229

"viewRootVar property" on page 230

### Related tasks

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a `pageHandler`" on page 232

"Changing the style class of a JSF component" on page 235

"Changing the style of a JSF component" on page 234

"Changing the target of a JSF link" on page 236

"Enabling or disabling JSF components" on page 237

"Setting event handlers for a JSF component"

"Setting JSF data table properties" on page 241

### Related reference

"Component tree access parts" on page 231

**Setting event handlers for a JSF component:** You can assign or remove a JavaScript function to a JSF component to serve as an event handler. In this context, an event handler is a JavaScript function that is called when a specific event happens on the page. For example, you can assign a function to a text input field using the `onClick` event handler. When the field is clicked in the browser, the function defined as the `onClick` event handler runs.

The JavaScript function used as an event handler must be available to the page, either in a <script> tag on the page itself or in a script file linked to the page. You can not use an EGL function as an event handler for a JSF component.

This task has the following prerequisites. For more information, see *Accessing a JSF component from a pageHandler*.

- Your EGL Web project must have support for the JSF component interface. See *Adding JSF component interface support to an EGL Web project*.
- The Faces JSP's page code file must have the following import statement:  

```
import com.ibm.egl.jsf.*
```
- You must declare a variable of type `UIViewRoot` within the pageHandler.
- You must specify the name of the `UIViewRoot` variable in the pageHandler property **viewRootVar**.

Follow these steps to assign or remove an event handler from a JSF component:

1. On a blank line inside a function in the pageHandler, press **Ctrl+Shift+Z**. The EGL Source Assistant window opens, displaying the JSF components on the page.
2. In the EGL Source Assistant window, select the JSF image component you want to access.
3. Click **OK**.

The EGL source assistant adds two lines of EGL code to the pageHandler. The first line defines an EGL variable of the type that matches the JSF component that you selected. The second line associates that variable with the JSF component. For example, the code to access a JSF input text component might look like this:

```
text1 HtmlInputText;  
text1 = myViewRoot.findComponent("form1:text1");
```

4. Using the EGL variable created by the source assistant, assign or remove the event handlers. For example, to assign the JavaScript function `myFunction()` as the `onClick` event handler for the text field, add this code:

```
text1.setOnClick("myFunction");
```

To remove an event handler from a JSF component, assign it a blank string as an event handler:

```
text1.setOnClick("");
```

### Related concepts

"JSF component tree" on page 229

"viewRootVar property" on page 230

### Related tasks

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a pageHandler" on page 232

"Changing the style class of a JSF component" on page 235

"Changing the style of a JSF component" on page 234

"Changing the target of a JSF link" on page 236

"Enabling or disabling JSF components" on page 237

"Setting the size of a JSF image" on page 238

"Setting JSF data table properties" on page 241

### Related reference

"Component tree access parts" on page 231

**Setting JSF data table properties:** You can change some of the properties of a JSF data table on a Faces JSP page with EGL code.

This task has the following prerequisites. For more information, see *Accessing a JSF component from a pageHandler*.

- Your EGL Web project must have support for the JSF component interface. See *Adding JSF component interface support to an EGL Web project*.
- The Faces JSP's page code file must have the following import statement:  

```
import com.ibm.egl.jsf.*
```
- You must declare a variable of type `UIViewRoot` within the pageHandler.
- You must specify the name of the `UIViewRoot` variable in the pageHandler property **viewRootVar**.

Follow these steps to assign or remove an event handler from a JSF component:

1. On a blank line inside a function in the pageHandler, press **Ctrl+Shift+Z**. The EGL Source Assistant window opens, displaying the JSF components on the page.
2. In the EGL Source Assistant window, select the JSF data table component you want to access.
3. Click **OK**.

The EGL source assistant adds two lines of EGL code to the pageHandler. The first line defines an EGL variable of the type that matches the JSF component that you selected. The second line associates that variable with the JSF component. For example, the code to access a JSF input text component might look like this:

```
table1 HtmlDataTable;  
table1 = viewRoot.findComponent("table1");
```

4. Using the EGL variable created by the source assistant, change the properties of the data table. For example, to change the table's `rowClasses` property to the style class `MyRowClass1`, add this code:

```
table1.setRowClasses("MyRowClass1");
```

To make the rows of the data table alternate between the two style classes `MyRowClass1` and `MyRowClass2`, add this code:

```
table1.setRowClasses("MyRowClass1, MyRowClass2");
```

### Related concepts

"JSF component tree" on page 229

"viewRootVar property" on page 230

### Related tasks

"Adding JSF component interface support to an EGL Web project" on page 230

"Accessing a JSF component from a pageHandler" on page 232

"Changing the style class of a JSF component" on page 235

"Changing the style of a JSF component" on page 234

"Changing the target of a JSF link" on page 236

"Enabling or disabling JSF components" on page 237

"Setting the size of a JSF image" on page 238

"Setting event handlers for a JSF component" on page 239

### Related reference

"Component tree access parts" on page 231

## Creating an EGL field and associating it with a Faces JSP

To create an EGL primitive field and associate it with a Faces JSP, do as follows:

1. Open a Faces JSP file in the Page Designer. To open a JSP file, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.

**Note:** You can access the related PageHandler by right clicking in the Design view (or Source view) and clicking **Edit Page Code**.

2. From the **Window** menu, select **Show View > Other > Basic > Palette**.
3. In the Palette view, click the **EGL** drawer to display the EGL data object types.
4. Drag **New Field** from the palette to the JSP. The Create a New EGL Data Field dialog is displayed.
5. Type a field name in the **Name** field.
6. Select the field type from the **Type** drop-down list and, if you need to specify the field's primitive characteristics (length and possibly decimals), type the information in the **Dimensions** text box. Default masks are used if you declare items of the following types:

- Date (mask *yyyymmdd*)
- Time (mask *hhmmss*)
- Timestamp (mask *yyyymmddhhmmss*)

If you wish to specify a DataItem part as the type, select **DataItem**, which is the last value in the list. In this case, the Select a DataItem part dialog is displayed, and you either select a DataItem part from the list or type the name, then click **OK**.

7. If you are creating an array of data items, select the **Array** check box and type an integer in the **Size** text box.
8. If you do not want to include the field on the page, clear the check box named **Add controls to display the EGL element on the Web page** and click **OK**. The field is now available in the Page Data view. You can add it to the JSP file later by dragging it from the Page Data view to the JSP.
9. If you want to include the fields in the JSP file, follow these additional steps:
10. Select the check box named **Add controls to display the EGL element on the Web page** and click **OK**. The Insert Control window opens.
11. In the Insert Control window, select the radio button that indicates your intended use of the field:
  - For output (**Displaying an existing record**)
  - For input or output (**Updating an existing record**)
  - For input (**Creating a new record**)

Your choice affects the types of controls that are available.

12. To change the field label, select the label that is displayed next to the field name, then type the new content.
13. To select a control type different from the one identified, select a type from the **Control Type** list.
14. If you click **Options**, the Options dialog is displayed, and the specific options that are available depend on whether you are using the field for input, for output, or for both. One option in any case is to include or exclude the JSF tag `<h:outputLabel>` around field labels.

When you complete your work in the Options dialog, click **OK**.

15. Click **Finish**.

#### Related reference

"Page Designer support for EGL" on page 221

"Primitive types" on page 34

## Associating an EGL record with a Faces JSP

To associate an EGL record with a Faces JSP, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP file opens in the Page Designer. Click the **Design** tab to access the Design view.

**Note:** You can access the related PageHandler by right clicking in the Design view (or Source view) and clicking **Edit Page Code**.

2. From the **Window** menu, select **Show View > Other > Basic > Palette**.
3. In the Palette view, click the **EGL** drawer to display the EGL data object types.
4. Drag **Record** from the palette to the JSP page. The Select a Record Part dialog is displayed.
5. Select a record from the list.
6. Specify the field name or accept the default value, which is the record-part name.
7. If you are declaring an array of records, select the **Array** check box and type an integer in the **Size** text box.
8. If you do not want to include the record on the page, clear the check box named **Add controls to display the EGL element on the Web page** and click **OK**. The record is now available in the Page Data view. You can add it to the JSP file later by dragging it from the Page Data view to the JSP.
9. If you want to include the fields in the JSP file, follow these additional steps:
10. Select the check box for **Add controls to display the EGL element on the Web page** and click **OK**. The Insert Control window opens.
11. In the Insert Control window, select the radio button that indicates your intended use of the field:
  - For output (**Displaying an existing record**)
  - For input or output (**Updating an existing record**)
  - For input (**Creating a new record**)

Your choice affects the types of controls that are available.

12. To change the order of fields, use the up and down arrows.
13. If you wish to select only a subset of the listed fields, click **None** and select the fields of interest. To select all fields instead, click **All**.
14. Do as follows for each field:
  - a. To exclude a field, clear the related check box. To include the field, ensure that the check box is selected.
  - b. To change the field label, select the label that is displayed next to the field name, then type the new content.
  - c. To select a control type different from the one identified (if possible), select from a list of types.
15. If you click **Options**, the Options dialog is displayed, and the specific options that are available depend on whether you are using fields for input, for output, or for both. One option in any case is to include or exclude the JSF tag `<h:outputLabel>` around field labels.

When you complete your work in the Options dialog, click **OK**.

16. Click **Finish**.

#### **Related concepts**

“Record parts” on page 135

#### **Related reference**

“Page Designer support for EGL” on page 221

## **Binding a JavaServer Faces command component to an EGL PageHandler**

To bind a JavaServer Faces command component (button or hypertext link) to an EGL PageHandler function, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View > Other > Basic > Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag a command component from the palette to the JSP. Command components have the word **Command** in the label. The component object is placed on the JSP.
5. Bind the an event handler to the command component using one of these methods:
  - To bind the component to an existing event handler, drag the event handler from the Actions node in the Page Data view to the component object on the JSP.
  - To create a new event handler that is bound to the component:
    - a. Right-click on the component and click **Edit Events** from the popup menu.
    - b. Using the Quick Edit view, enter the EGL code for the event handler. For details on using the Quick Edit view, see *Using Quick Edit view for PageHandler code*.

The event handler is visible in the Page Data view and a corresponding function is added to the PageHandler. For details, see *PageHandler part in EGL source format*.

#### **Related concepts**

“PageHandler” on page 223

#### **Related tasks**

“Creating an EGL pageHandler part” on page 221

“Using the Quick Edit view for PageHandler code”

#### **Related reference**

“Page Designer support for EGL” on page 221

“PageHandler part in EGL source format” on page 785

## **Using the Quick Edit view for PageHandler code**

The Quick Edit view allows you to maintain EGL PageHandler code for JSP server events without opening the PageHandler file. To use the Quick Edit view, do as follows:



1. Open a JSP file in the Page Designer. If you do not have a file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. Right-click in the Page Designer, then select **Edit Events**. The Quick Edit view opens.
3. Follow these steps to maintain PageHandler functions for command components:
  - a. Select a command component in the JSP.
  - b. If the command component already has a PageHandler function associated with it, the function displays in the script editor (right pane) of the Quick Edit view. Any changes you make to the code are reflected in the PageHandler.
  - c. To create a PageHandler function for the selected command component, click **Command** in the event pane (left pane) of the Quick Edit view, then click in the script editor (right pane) of the Quick Edit view. The function displays. Type the PageHandler code for the function.
4. Follow these steps to maintain the onPageLoad function:
  - a. Click inside the JSP.
  - b. Click **onPageLoad** in the event pane (left pane) of the Quick Edit view.
  - c. The onPageLoad function displays in the script editor (right pane) of the Quick Edit view. Any changes you make to the code are reflected in the PageHandler.

#### Related concepts

"PageHandler" on page 223

#### Related reference

"PageHandler part in EGL source format" on page 785

## Binding a JavaServer Faces input or output component to an EGL PageHandler

To bind a JavaServer Faces input or output component to an existing EGL PageHandler data area, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View > Other > Basic > Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag an input or output component from the palette to the JSP. Input and output components have the words **Input** and **Output** in the labels. The component object is placed on the JSP.
5. To bind the component to an existing PageHandler data area, do one of the following:
  - Drag the data area from the Page Data view to the component object on the JSP.
  - Select the component object on the JSP, then right-click the data area in the Page Data view and select **Bind to 'component name'**.
  - Select the component object on the JSP. Click the button next to the **Value** field of the Properties view, then select a data area from the Select Page Data Object list and click **OK**.



#### Related concepts

"PageHandler" on page 223

#### Related tasks

"Creating an EGL pageHandler part" on page 221

#### Related reference

"Page Designer support for EGL" on page 221

"PageHandler part in EGL source format" on page 785

## Binding a JavaServer Faces check box component to an EGL PageHandler

A JavaServer Faces check box component is unique in that the data area to which it is bound must have the item property **isBoolean** (formerly the **boolean** property) set to *yes*. Examples of boolean data area declarations are as follows:

```
DataItem CharacterBooleanItem char(1)
{
    value = "N",
    isBoolean = yes
}
end

DataItem NumericBooleanItem smallInt
{
    value = "0",
    isBoolean = yes
}
end
```

To bind a JavaServer Faces check box component to an existing EGL PageHandler data area, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View > Other > Basic > Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag a check box component from the palette to the JSP. The component object is placed on the JSP.
5. To bind the component to an existing PageHandler data area, do one of the following:
  - Drag the data area from the Page Data view to the component object on the JSP.
  - Select the component object on the JSP, then right-click the data area in the Page Data view and select **Bind to 'component name'**.
  - Select the component object on the JSP. Click the button next to the **Value** field in the Properties view, then select a data area from the Select Page Data Object list and click **OK**.

#### Related concepts

"PageHandler" on page 223

#### Related tasks

"Creating an EGL pageHandler part" on page 221

### Related reference

“Page Designer support for EGL” on page 221

“PageHandler part in EGL source format” on page 785

## Binding a JavaServer Faces single-selection component to an EGL PageHandler

A single-selection component allows a user to make one selection from a list of values. The user’s selection is stored in a PageHandler data area. Radio buttons, single-select list boxes, and combo boxes are single-selection JavaServer Faces components.

A binding is a relationship between the component and a data area. The data area must be declared in the PageHandler before you can bind a component to it. A single-selection component needs two different kinds of bindings:

- A binding to one or more data areas that contain the values from which the user can make a selection
- A binding to a data area that will receive the user’s selection

To bind a JavaServer Faces single-selection component to existing EGL PageHandler data areas, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View > Other > Basic > Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag a single-selection component from the palette to the JSP. The component object is placed on the JSP.
5. To bind the component to one or more PageHandler data areas that contain the values you want to display to the user, do one of the following procedures:
  - You can bind the component to individual PageHandler data areas, each of which contains one list item. Perform the following procedure for each data area:
    - a. Select the object component on the JSP.
    - b. In the Properties view, click **Add Choice**. The Name and Value fields are populated with default values.
    - c. Click the **Name** field, then type the text that you want to display to the user.
    - d. Click the **Value** field, then click the button next to the **Value** field. Select an individual data area from the Select Page Data Object list and click **OK**. This data area holds the value that will be moved to the receiving data area.
  - You can bind the component to a PageHandler array data area that contains the values you want to display to the user. Perform the following procedure to bind the component to an array data area:
    - a. Select the object component on the JSP.
    - b. In the Properties view, click **Add Set of Choices**. The Name and Value fields are populated with default values.
    - c. Click the **Value** field, then click the button next to the **Value** field. Select an array data area from the Page Data Object list and click **OK**. The values in the array data area are the values that will be displayed to the

user. The properties described later determine whether the values in the array data area or their equivalent index values will be moved to the receiving data area.

6. If you are using an array data area to supply the values displayed to the user, you must define the receiving data area with two properties: **selectFromListItem** and **selectType**. The **selectFromListItem** property points to the array that holds the list items. The **selectType** property indicates whether the receiving data area is to be populated with a text value or an index value. Examples of receiving data areas are as follows:

```
colorSelected char(10)
{selectFromListItem = "colorsArray",
 selectType = value};

colorSelectIdx smallInt
{selectFromListItem = "colorsArray",
 selectType = index};
```

7. To bind the component to a PageHandler data area that will receive the user's selection, do one of the following:
  - Drag the data area from the Page Data view to the component object on the JSP.
  - Select the component object on the JSP, then right-click the data area in the Page Data view and select **Bind to 'component name'**.
  - Select the component object on the JSP. Click the button next to the **Value** field in the Properties view, then select a data area from the Select Page Data Object list and click **OK**.

#### Related concepts

"PageHandler" on page 223

#### Related tasks

"Creating an EGL pageHandler part" on page 221

#### Related reference

"Page Designer support for EGL" on page 221

"PageHandler part in EGL source format" on page 785

## Binding a JavaServer Faces multiple-selection component to an EGL PageHandler

A multiple-selection component allows a user to make one or more selections from a list of values. The user's selections are stored in a PageHandler array data area. Check box groups and multiple-select list boxes are multiple-selection JavaServer Faces components.

A binding is a relationship between the component and a data area. The data area must be declared in the PageHandler before you can bind a component to it. A multiple-selection component needs two different kinds of bindings:

- A binding to one or more data areas that contain the values from which the user can make a selection
- A binding to an array data area that will receive the user's selections

To bind a JavaServer Faces multiple-selection component to existing EGL PageHandler data areas, do as follows:

1. Open a Faces JSP file in the Page Designer. If you do not have a JSP file open, double-click on the JSP file in the Project Explorer. The JSP opens in the Page Designer. Click the **Design** tab to access the Design view.
2. From the **Window** menu, select **Show View > Other > Basic > Palette**.
3. In the Palette view, click the **Faces Components** drawer to display the Faces Components object types.
4. Drag a multiple-selection component from the palette to the JSP. The component object is placed on the JSP.
5. To bind the component to one or more PageHandler data areas that contain the values you want to display to the user, do one of the following procedures:
  - You can bind the component to individual PageHandler data areas, each of which contains one list item. Perform the following procedure for each data area:
    - a. Select the object component on the JSP.
    - b. In the Properties view, click **Add Choice**. The Name and Value fields are populated with default values.
    - c. Click the **Name** field, then type the text that you want to display to the user.
    - d. Click the **Value** field, then click the button next to the **Value** field. Select an individual data area from the Select Page Data Object list and click **OK**. This data area holds the value that will be moved to the receiving data area.
  - You can bind the component to a PageHandler array data area that contains the values you want to display to the user. Perform the following procedure to bind the component to an array data area:
    - a. Select the object component on the JSP.
    - b. In the Properties view, click **Add Set of Choices**. The Name and Value fields are populated with default values.
    - c. Click the **Value** field, then click the button next to the **Value** field. Select an array data area from the Select Page Data Object list and click **OK**. The values in the array data area are the values that will be displayed to the user. The properties described later determine whether the values in the array data area or their equivalent index values will be moved to the receiving data area.
6. If you are using an array data area to supply the values displayed to the user, you must define the receiving data area with two properties: **selectFromListItem** and **selectType**. The **selectFromListItem** property points to the array that holds the list items. The **selectType** property indicates whether the receiving data area is to be populated with a text value or an index value. Examples of receiving data areas are as follows:
 

```
colorSelected char(10)
{selectFromListItem = "colorsArray",
 selectType = value};

colorSelectIdx smallInt
{selectFromListItem = "colorsArray",
 selectType = index};
```
7. To bind the component to a PageHandler array data area that will receive the user's selections, do as follows:
  - a. Select the component object on the JSP
  - b. Click the button next to the **Value** field of the Properties view
  - c. Select a data area from the Select Page Data Object list
  - d. Click **OK**

**Related concepts**

“PageHandler” on page 223

**Related tasks**

“Creating an EGL pageHandler part” on page 221

**Related reference**

“Page Designer support for EGL” on page 221

“PageHandler part in EGL source format” on page 785

---

## Segmentation in Web transactions

Segmentation concerns how a program interacts with its environment before issuing a **converse** statement.

If a program issues a **converse** statement to present a Web page, the runtime behavior depends on whether the code is generated as a Java program or as a CICS COBOL program:

- The EGL-generated Java program is *non-segmented*, which means that the program remains in memory during the period when the user considers how to respond.

The Java program acts as follows before presenting the Web page:

- Commits databases and other recoverable resources.
- Releases locks, as well as file and database positions.
- Refreshes single-user EGL tables so that their values become the same as when the program began.
- Refreshes system variables so that their values become the same as when the program began, except for a subset of variables whose values are saved. The save status is noted in the page for each variable.

- The EGL-generated CICS COBOL program is *segmented*, which means that the program leaves memory after issuing the **converse** statement and is returned to memory when the user responds. For details, see *Behavior of a segmented program on CICS*.

You cannot control the segmentation status of an EGL-generated program in a Web application, as you can for a program in a text application.

**Related concepts**

“Program part” on page 148

“Developing Web transactions in EGL” on page 165

---

## Creating EGL Reports

---

### EGL reports overview

EGL can produce reports based on the JasperReports open-source, Java-based reporting library. For details on that library, see the following Web site:

<http://jasperreports.sourceforge.net>

EGL does not provide a mechanism for report layout. You must do this:

- Import a JasperReports design file (extension *jasper*); or
- Use a text editor or specialized tool to create a JasperReports XML source file (extension *jrxml*). The EGL runtime compiles that source into a .jasper design file automatically; for details on this process, see *Creating the report design file*.

Here are two specialized tools for creating a design file:

- JasperAssistant, described on this Web site:

<http://www.jasperassistant.com>

- iReport, described on this Web site:

<http://ireport.sourceforge.net>

The EGL report driver (which you write to drive report production) identifies the JasperReports design file (extension *jasper*) and relies on that design file for a number of formatting definitions:

- The design file defines fonts, headers, footers, the fields to appear in the report, positions of those fields, subtotals from those fields, and other basic components of the report.
- If you use a database connection as the source of the report, the design file will probably contain your SQL query.
- The design file can create nested subreports to provide additional data for each line item in the report. See *Creating Subreports*.

At run time, your EGL report driver sets up basic parameters, then hands control to the JasperReports engine. The JasperReports engine creates an intermediate file called the destination file (extension *jrprint*) and fills this file with report data. Based on the specifications in your driver file, the JasperReports engine then formats the report data for one or more exported files. Exported file formats can be .pdf, .html, .xml, .txt, and/or .csv.

If you also code an EGL handler of type `JasperReport`, the finished report can reflect events that occurred as your EGL report driver filled the report with data. For example, you can produce dynamic report content by comparing report subtotals with outside information like commission structures or insurance reimbursements.

When you write EGL code that interacts with a report, you do these things:

- Create variables that are based on the predefined parts `Report` and `ReportData`.
- Interact with the design file by invoking functions from the system library `ReportLib`, using those variables as arguments in the function invocations

### Related concepts

"EGL report creation process overview"

### Related tasks

"Creating the report design file" on page 254

"Creating subreports" on page 263

### Related reference

"EGL library ReportLib" on page 990

"EGL report handler" on page 264

---

## EGL report creation process overview

To create a report, complete the three processes described below:

- Create a JasperReports design file (required)
- Write code to drive a report (required)
- Create a report handler (optional)

EGL does not require that you perform these tasks in a particular order (with the exceptions described in the "Code interrelationships between the report handler and the XML source" paragraphs in step 2 below). If you want a report handler, you can create it before you create a JasperReports design file, or you can create a report handler and the XML source for the design file simultaneously

You cannot create a report if you do not have a compiled JasperReports design file and the code for driving the report.

Use EGL ReportLib functions to write report-invocation code in your EGL project. You can use the EGL Program Part wizard when creating report-invocation code.

To create and fill a report after you create an XML source design file, a report handler (if you want to use one), and report-invocation code, you complete these steps:

1. Build the EGL project by selecting **Project > Build All**.

EGL automatically generates Java code from the EGL report handler (if you created one) and from the EGL report driver. (EGL compiles the XML design file into a .jasper file automatically each time you save it. You can force EGL to recompile XML design file by selecting **Project > Clean**.)

2. Run the EGL program that has the report invocation code.

As the EGL program runs, the JasperReports program used by EGL first fills and stores a .jrprint file. This file uses an intermediate file format that the program will then export into the final report format (.pdf, .html, .xml, .txt, or .csv).

The JasperReports program saves the filled report in the location specified by *reportDestinationFileName* in the report-invocation code. The program can reuse one .jrprint file for multiple exports by pairing appropriate file locations with *exportReport()* functions in the report driver code. For example, the following code creates a report in .pdf format:

```
myReport.ReportExportFile = "C:\\temp\\my_report.pdf";  
reportLib.exportReport(myReport, ExportFormat.pdf);
```

EGL does not automatically refresh exported reports. If you change the report design or if data changes, you must run the report driver again.



### Related concepts

“EGL reports overview” on page 251

### Related tasks

“Migrating EGL code to EGL V6.0 iFix” on page 103

“Creating the report design file” on page 254

“Exporting reports” on page 274

“Generating files for and running a report” on page 273

“Using report templates” on page 259

“Using the EGL templates with content assist” on page 131

“Writing code to drive a report” on page 257

### Related Reference

“EGL library ReportLib” on page 990

“EGL report handler” on page 264

## JasperReports design file

Create or import a JasperReports design file (extension *jasper*). To create the design file, put together an XML source file specifying layout information for the report. You can create this source file in one of two ways:

- Use a third-party JasperReports design tool (like JasperAssistant or iReport).
- Use a text editor to write Jasper XML design information into a new text file.

The XML source file should have a *.jrxml* extension. Be sure to save the XML design document in the same EGL package that will contain the report driver code files and the EGL report handler (if you use one).

The EGL runtime will automatically compile a valid XML source file into a JasperReports design file each time you save the source to disk (provided you have the *javac* compiler on your execution path). If you do not create a new *.jrxml* file, you must import a previously compiled *.jasper* file.

## Report driver

### Optional report handler

If you want to use a report handler, which provides the logic for handling events during the filling of the report, you can create one in either of the following ways:

- Use the EGL report handler wizard to specify information for the report handler.
- Create a new EGL source file and either insert a handler using the report handler template or manually enter the handler code.

**Code interrelationships between the report handler and the XML source.** In the *.jrxml* file, you can specify the *scriptletClass* that references the report handler file that the EGL report handler generates. Be aware of the following:

- If the *.jrxml* file uses Java code that a report handler produces, you must generate the report handler before you create the *.jrxml* file.
- If you change a report handler, you must recompile the *.jrxml* file.
- If you need to resolve any compilation errors in the *.jrxml* file, you must modify the *.jrxml* file and save it. If you need to recompile the *.jasper* file after making changes to a report handler, you can force a recompile by selecting **Project > Clean**.



---

## Creating the report design file

EGL works with the JasperReports open-source, Java-based reporting library to create reports. Unless you import a working design file (extension *jasper*), you will need to create or modify an XML source for the design file. For more detailed information than you will find in this topic, see the following Web site:

<http://jasperreports.sourceforge.net>

You can use a file with the .xml extension for your source, though this could slow your compilation. JasperReports recommends the .jrxml extension.

To add a design document to a package, follow these steps:

1. Create a design document in one of the following ways:
  - Use a third-party JasperReports design tool (like JasperAssistant or iReport). Make sure the file you create has a .jrxml extension.
  - Use a text editor to write JasperReports XML source information into a new text file and save the file as a .jrxml file.
2. Place the XML design document in the same EGL package as your report driver file and optional EGL report handler.
3. Customize the XML source file to use one of the following data sources:
  - When you have a fairly simple, straightforward database query, create a report of type **DataSource.databaseConnection**. Include your SQL query in the XML design file source. The EGL report driver passes your connection information to JasperReports.
  - When you need to perform complex database operations, or need to build your SQL statement dynamically, create a report of type **DataSource.sqlStatement**. The EGL report driver includes your SQL query and passes the result set to JasperReports.
  - When your data comes from somewhere other than a database, create a report of type **DataSource.reportData**. The EGL report driver passes complete report data to JasperReports; no connection information is necessary.

The following sections offer specifics for the different types of XML source files. This information covers very simple cases; for more complex examples see either the JasperReports Web site mentioned earlier or the documentation for your design tool (if you decide to use one).

### EGL source files of type **DataSource.databaseConnection**

Include connection information in your EGL source file and include your SQL query in the XML design file source. Here is the syntax for a very simple case:

```
<queryString><![CDATA[SELECT * FROM Table_Name]]></queryString>
```

*Table\_Name*

Name of a table from your database

Define the specific fields (tied to columns in the SQL result set) you want to use:

```
<field name="Field_Name" class="java.lang.class_type"></field>
```

*Field\_Name*

A column name in the result set from the query in your design file. The field

names must conform to Java variable name conventions. You can alias column names within your SQL statement to handle duplicate names, illegal characters (such as "."), or other conflicts.

#### *Class\_Type*

A java.lang class such as Integer or String, identifying the type of data to which *Field\_Name* refers

Place the fields on the report with a TextFieldExpression tag:

```
<textFieldExpression class="java.lang.class_type">${Field_Name}]]&gt;&lt;/textFieldExpression&gt;</pre></div><div data-bbox="205 239 762 260" data-label="Section-Header"><h2>EGL source files of type DataSource.sqlStatement</h2></div><div data-bbox="276 263 906 338" data-label="Text"><p>Place your SQL statement in the EGL report driver file. You can either specify your connection explicitly in your report driver or use the default connection in your build descriptor. In the XML design file, define the specific fields you want to print on the report. The field names refer to column names in your SQL statement's result set:</p></div><div data-bbox="276 341 739 357" data-label="Text"><pre>&lt;field name="Field_Name" class="java.lang.class_type"&gt;&lt;/field&gt;</pre></div><div data-bbox="276 363 364 379" data-label="Section-Header"><h4><i>Field_Name</i></h4></div><div data-bbox="307 379 876 440" data-label="Text"><p>A column name in the result set that was created by the query in your EGL report driver. The field names must conform to Java variable name conventions. You can alias the column names within your SQL statement if necessary.</p></div><div data-bbox="276 446 360 463" data-label="Section-Header"><h4><i>Class_Type</i></h4></div><div data-bbox="307 462 855 492" data-label="Text"><p>A java.lang class such as Integer or String, identifying the type of data to which <i>Field_Name</i> refers</p></div><div data-bbox="276 506 732 523" data-label="Text"><p>Place the fields on the report with a TextFieldExpression tag:</p></div><div data-bbox="276 525 1000 541" data-label="Text"><pre>&lt;textFieldExpression class="java.lang.class_type"&gt;<![CDATA[${Field_Name}]]&gt;&lt;/textFieldExpression&gt;</pre></div><div data-bbox="205 554 734 575" data-label="Section-Header"><h2>EGL source files of type DataSource.reportData</h2></div><div data-bbox="276 577 881 639" data-label="Text"><p>Here you do not use a database, so you do not need a connection or SQL statement, either in your XML source or in the EGL report driver. Instead define the specific fields you want to use from the records that you create in the EGL report driver:</p></div><div data-bbox="276 641 739 656" data-label="Text"><pre>&lt;field name="Field_Name" class="java.lang.class_type"&gt;&lt;/field&gt;</pre></div><div data-bbox="276 662 364 678" data-label="Section-Header"><h4><i>Field_Name</i></h4></div><div data-bbox="307 678 832 694" data-label="Text"><p>The name of a field exactly as you specified it in your EGL source file</p></div><div data-bbox="276 700 360 717" data-label="Section-Header"><h4><i>Class_Type</i></h4></div><div data-bbox="307 716 855 746" data-label="Text"><p>A java.lang class such as Integer or String, identifying the type of data to which <i>Field_Name</i> refers</p></div><div data-bbox="276 761 732 777" data-label="Text"><p>Place the fields on the report with a TextFieldExpression tag:</p></div><div data-bbox="276 779 1000 795" data-label="Text"><pre>&lt;textFieldExpression class="java.lang.class_type"&gt;<![CDATA[${Field_Name}]]&gt;&lt;/textFieldExpression&gt;</pre></div><div data-bbox="205 807 625 828" data-label="Section-Header"><h2>Compiling the XML design file source</h2></div><div data-bbox="276 830 892 863" data-label="Text"><p>EGL automatically compiles any .jrxml file it finds in the package directory in the EGL source folder, providing these conditions are true:</p></div><div data-bbox="276 864 636 919" data-label="List-Group"><ul><li>• The .jrxml file has changed</li><li>• The .jrxml file is free of errors</li><li>• The javac compiler is on your execution path</li></ul></div><div data-bbox="715 937 906 955" data-label="Page-Footer"><p>Creating EGL Reports 255</p></div>
```

EGL places the compiled .jasper file in the JavaSource\*package\_name* directory that is parallel to EGLSource\*package\_name*. When you successfully generate your EGL report driver, the product places a linked copy of the .jasper file in the parallel bin\*package\_name* directory. You can force the .jasper file creation and copy by selecting **Project > Build All** or **Project > Clean**.

If you are creating an XML design document and a report handler simultaneously, see *EGL report creation process overview* for guidelines. For an example that shows how an XML design document gets a report data record from the report handler, see *Creating an EGL report handler*.

**Related concepts**“EGL reports overview” on page 251  
“EGL report creation process overview” on page 252

**Related tasks**

“Migrating EGL code to EGL V6.0 iFix” on page 103  
“Writing code to drive a report” on page 257

**Related reference**

“EGL library ReportLib” on page 990  
“Data types in XML design documents”

---

## Data types in XML design documents

In XML report design documents, data types are described as Java data types. If you invoke EGL report handler functions from the design document, the invocation should use the Java data type that corresponds to the applicable EGL primitive type. You must also declare the data that the report handler function returns to the XML design file in terms of Java data types.

The following table shows how EGL primitive types map to Java data types. JasperReports documentation contains information on the Java data types that you can use.

EGL primitive type	Java data type
bigint	java.lang.Long
bin	java.math.BigDecimal
blob	
char	java.lang.String
clob	
date	java.util.Date
dbchar	java.lang.String
decimal	java.math.BigDecimal
decimalfloat	java.lang.Double
float	java.lang.Float
hex	java.lang.Byte
int	java.lang.Integer
interval	java.lang.String
mbchar	java.lang.String
money	java.math.BigDecimal

EGL primitive type	Java data type
numc	java.math.BigDecimal
pacf	java.math.BigDecimal
smallfloat	java.lang.Float
smallint	java.lang.Short
string	java.lang.String
time	java.sql.Time
timestamp	java.sql.Timestamp
unicode	java.lang.String

### Related concepts

“EGL reports overview” on page 251

“EGL report creation process overview” on page 252

### Related tasks

“Creating the report design file” on page 254

### Related reference

“EGL library ReportLib” on page 990

“EGL report handler” on page 264

“Report and ReportData parts” on page 262

---

## Writing code to drive a report

Use the New EGL Part wizard to create a new EGL basic program that uses the report library to run reports.

To create this report driver, follow these steps:

1. Choose **File > New > Program** and then select the folder that will contain the EGL source file.
2. Select a package.
3. Specify a file name for the source file, select the *BasicProgram* type and click **Finish**.
4. Open the new file in the EGL editor.
5. In the main() function, just after the program line, type **jas** followed by **Ctrl+space** to insert code for the report driver.
6. In the pop-up window offering data-source connection types and code, select one of the data-source connection types.
7. You can modify existing code or add your own code. If you modify the code, insert specific values for the placeholders the editor provided. These placeholders might include *reportDesignFileName*, *reportDestinationFileName*, *exportReportFile*, *alias*, *databaseName*, *userid*, *password*, and *connectionName*.

### Code showing report-invocation information

The following code shows report-invocation information for a report where you specify the SQL statement in the EGL source file, using the default connection in your build descriptor and exporting the file in PDF:

```

myReport      Report;
myReportData  ReportData;

//Initialize Report file locations
myReport.reportDesignFile =
"c:\\workspace\\report_project\\bin\\report_package\\myReport.jasper";
myReport.reportDestinationFile = "c:\\temp\\myReport.jrprint";

//Get the report data via a SQL statement
myReportData.sqlStatement = "Select * From myTable";
myReport.reportData = myReportData;

//Fill the report with data
ReportLib.fillReport(myReport, DataSource.sqlStatement);

//Export the report in pdf
myReport.reportExportFile = "c:\\temp\\myReport.pdf";
ReportLib.exportReport(myReport, ExportFormat.pdf);

```

Because the backslash character ("\") is used in escape sequences, you must use a double backslash in path names.

Code	Explanation
myReport Report;	A report library record declaration
myReportData ReportData;	A report library data record declaration
myReport.reportDesignFile = "c:\\workspace\\..."	Full path to the .jasper design file
myReport.reportDestinationFile ="c:\\temp\\myReport.jrprint";	Full path to the intermediate (destination) work file
myReport.sqlStatement = "Select * From myTable";	The SQL statement used to provide data to the report
myReport.reportData = myReportData;	Assigns the data the variable of type Report
ReportLib.fillReport(myReport, DataSource.sqlStatement );	Defines the data source for the report to be an SQL statement
myReport.reportExportFile = "c:\\temp\\myReport.pdf	Full path to the file to be created/used for PDF output
ReportLib.exportReport(myReport, ExportFormat.pdf);	Specifies the report export format

### Related concepts

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

### Related tasks

"Migrating EGL code to EGL V6.0 iFix" on page 103

"Creating the report design file" on page 254

"Using report templates" on page 259

### Related reference

"EGL report handler" on page 264

"EGL library ReportLib" on page 990

"Sample code for EGL report-driver functions" on page 260

---

## Using report templates

EGL provides code that you can add to your program with a few keystrokes. That code can give you a template for writing a report handler or can provide most of the statements you will need to fill and export a report through Jasper Reports.

In the second case, you can choose from any of these data sources for the report:

- Database connection
- Custom report data
- Result object from a specified SQL query

To add a report handler template to your source file, follow these steps:

1. Open a new file in your text editor.
2. Type **handler**, then press **Ctrl+space**
3. The editor will replace the word "handler" with template code. Work through the code and add statements for the functions you wish to use. For more information, including code examples, see *Creating an EGL report handler*.

To add JasperReport code to your source file, follow these steps:

1. Move your cursor to a new line within the main() section of your program.
2. Type **jas**, then press **Ctrl+space**
3. The editor will offer you a menu with options for database, custom, or SQL data; select the appropriate template.
4. The editor will replace the letters "jas" with code; use the **Tab** key to move to the fields you need to change.

You can edit the templates themselves by following these steps:

1. Select **Window > Preferences**.
2. When a list of preferences is displayed, expand **EGL**.
3. Expand **Editor** and select **Templates**.
4. Scroll through the list of templates and select a template. For example, select **handler** to display the report handler template.
5. Click **Edit**.
6. Change the template to meet your needs.
7. Click **Apply** and then **OK** to save your changes.

### Related concepts

"EGL report creation process overview" on page 252

"EGL reports overview" on page 251

### Related tasks

"Migrating EGL code to EGL V6.0 iFix" on page 103

"Setting preferences for templates" on page 119

"Using the EGL templates with content assist" on page 131

"Writing code to drive a report" on page 257

### Related Reference

"EGL library ReportLib" on page 990

"EGL report handler" on page 264

"Sample code for EGL report-driver functions" on page 260

---

## Sample code for EGL report-driver functions

Here are code snippets that show how to fill a report using three different data sources:

- A database connection
- A data record
- A result set from an SQL statement

The following code snippet shows how you might fill a report using a database connection as the data source (the SQL statement is in the .jasper design file):

```
//Variable declaration
myReport      Report;
myReportData  ReportData;

//Function containing report invocation code
function makeReport()
  //Initialize Report file locations
  myReport.reportDesignFile =
    "c:\\workspace\\report_project\\bin\\report_package\\myReport.jasper";
  myReport.reportDestinationFile = "c:\\temp\\myReport.jrprint";

  //Get the report data via a connection
  sysLib.defineDatabaseAlias("alias", "connectionName");
  sysLib.connect("alias", "userid", "password");
  myReportData.connectionName = "alias";
  myReport.reportData = myReportData;

  //Fill the report with data
  reportLib.fillReport(myReport, DataSource.databaseConnection);

  //Export the report in PDF format
  myReport.reportExportFile = "c:\\temp\\reportDesignFileName.pdf";
  reportLib.exportReport(myReport, ExportFormat.pdf);
end
```

The following code snippet shows how you might fill a report using your own internal record as the data source. First, you'll need to define a record like the following somewhere outside the program:

```
Record myRecord type BasicRecord
  author String;
  description String;
  title String;
end
```

Next you'll write the function to fill and drive the report:

```
//Variable declarations
myReport      Report;
myReportData  ReportData;

recArray      myRecord[];
recArrayElement myRecord;

//Function containing the report driving code
function makeReport()
  //Initialize report file locations
  myReport.reportDesignFile =
    "c:\\workspace\\report_project\\bin\\report_package\\myReport.jasper";
  myReport.reportDestinationFile = "c:\\temp\\myReport.jrprint";

  //Get the report data
```

```

populateReportData();
myReport.reportData = myReportData;

//Fill the report with data
reportLib.fillReport(myReport, DataSource.reportData);

//Export the report in HTML format
myReport.reportExportFile = "c:\\temp\\myReport.html";
reportLib.exportReport(myReport, ExportFormat.html);
end

function populateReportData()
recArrayElement.author="Jane Austen";
recArrayElement.title="Northanger Abbey";
recArrayElement.description = "British Novel";
recArray.appendElement(recArrayElement);

recArrayElement.author = "Jane Austen";
recArrayElement.title="Emma";
recArrayElement.description = "British Novel";
recArray.appendElement(recArrayElement);

recArrayElement.author = "Charles Dickens";
recArrayElement.title="Our Mutual Friend";
recArrayElement.description = "British Novel";
recArray.appendElement(recArrayElement);

recArrayElement.author = "Gustave Flaubert";
recArrayElement.title="Madame Bovary";
recArrayElement.description = "French Novel";
recArray.appendElement(recArrayElement);

recArrayElement.author = "M. Lermontov";
recArrayElement.title="Hero of Our Time";
recArrayElement.description = "Russian Novel";
recArray.appendElement(recArrayElement);
end

```

The following code snippet shows how you might fill a report using an SQL statement as the data source. This example assumes a default database connection in your program properties file:

```

//Variable declaration
myReport      Report;
myReportData  ReportData;

//Function containing report driving code
function makeReport()
//Initialize Report file locations
myReport.reportDesignFile =
    "c:\\workspace\\report_project\\bin\\report_package\\myReport.jasper";
myReport.reportDestinationFile = "c:\\temp\\myReport.jrprint";

//Get the report data via SQL statement
myReportData.sqlStatement = "SELECT * FROM dataBaseTable";
myReport.reportData = myReportData;

//Fill the report with data
reportLib.fillReport(myReport, DataSource.sqlStatement);

//Export the report in text format
myReport.reportExportFile = "c:\\temp\\myReport.txt";
reportLib.exportReport(myReport, ExportFormat.text);
end

```



**Related concepts**

“EGL report creation process overview” on page 252

“EGL reports overview” on page 251

**Related tasks**

“Writing code to drive a report” on page 257

**Related reference**

“Report and ReportData parts”

“EGL report handler” on page 264

“EGL library ReportLib” on page 990

---

## Report and ReportData parts

In creating EGL reports, you will use variables of type *Report* and *ReportData*.

The variable of type *Report* contains information specific to a report. The variable contains these fields:

Field	Explanation	Data Type
<i>reportDesignFile</i>	Full path to the report design file, which is a compiled XML file with a .jasper extension	String
<i>reportDestinationFile</i>	Full path to the intermediate .jrprint file	String
<i>reportExportFile</i>	Full path to the final.xml, .pdf, .html, .txt, or .csv file	String
<i>reportData</i>	Reference to the actual data for the report	ReportData

The variable of type *ReportData* describes the data to be used in a report. The variable contains these fields:

Field	Explanation	Data type
<i>connectionName</i>	Alias for a database connection that will provide data for the report (DataSource.databaseConnection)	String
<i>sqlStatement</i>	The SQL statement that provides data to the report (DataSource.sqlStatement)	String
<i>Data</i>	Reference to a dynamic array of records (DataSource.reportData)	Any (This is the EGL <i>Any</i> type.)

**Related concepts**

“EGL report creation process overview” on page 252

“EGL reports overview” on page 251

**Related reference**

“EGL library ReportLib” on page 990

## Creating subreports

In the simplest case, the subreport capability of JasperReports allows you to print associated data for each line item in a report—from the same table, from another table in the same database, or from another data source altogether. More complex cases are beyond the scope of this topic, and you should refer to JasperReports documentation for such information.

In complex cases you will probably need a report handler to provide data for the subreport; see *Creating an EGL report handler*. In some simple cases, you can add a subreport in two simple steps:

1. Add subreport code to the design file for the main report.
2. Create one or more design files for the subreport(s).

Here is an elementary example of how you might do this. Given a report that prints all the customers from the table CUSTOMER, you can add a subreport that shows all invoices for each customer, drawn from the table ORDERS.

1. This is a section of code from the main report design file. This code was originally written to print one line for each customer in the table CUSTOMER. Add the code shown in bold in the following example:

```
<queryString><![CDATA[SELECT * FROM ADMINISTRATOR.CUSTOMER]]></queryString>
<field name="CUST_NO" class="java.lang.Integer">
</field>
<field name="CUST_NAME" class="java.lang.String">
</field>
<detail>
<band height="100">
  <subreport>
    <reportElement positionType="Float" mode="Opaque" x="0" y="31" width="709" height="12" isR
    <subreportParameter name="CURRENT_CUST">
      <subreportParameterExpression><![CDATA[${CUST_NO}]]></subreportParameterExpression>
    </subreportParameter>
    <connectionExpression>
      <![CDATA[${REPORT_CONNECTION}]]>
    </connectionExpression>
    <subreportExpression class="java.lang.String"><![CDATA[new String("C:\\workspace\\report_p
    </subreport>
  <textField>
    <reportElement positionType="Float" x="57" y="11" width="304" height="20"/>
    <textElement/>
    <textFieldExpression class="java.lang.String"><![CDATA[${CUST_NO} + " " + ${CUST_NAME}]]
    </textField>
  </band>
</detail>
```

The `<subreport>` tag gives JasperReports the information it needs to run the subreport:

- positioning information (the same parameter you will find in the main report)
- parameters you wish to pass to the subreport—in this case, the number of the current customer, which you will need in the SQL SELECT statement in the subreport
- connection information, because the report driver for this report specifies a data source of `DataSource.databaseConnection`

- the location of a compiled report design file for the subreport—in this case, my\_subreport.jasper
2. The subreport design file is not different in kind than any other .jasper design file. Include the following crucial code in that file:

```
<parameter name="CURRENT_CUST" class="java.lang.Integer"/>
<queryString><![CDATA[SELECT * FROM ADMINISTRATOR.ORDERS WHERE CUST_NO = $P{CURRENT_CUST}]]></queryString>
<field name="CUST_NO" class="java.lang.Integer">
</field>
<field name="INVOICE_NO" class="java.lang.Integer">
</field>
<field name="ORDER_TOTAL" class="java.lang.Float">
</field>
<detail>
  <band height="100">
    <textField>
      <reportElement positionType="Float" x="50" y="10" width="300" height="20"/>
      <textElement/>
      <textFieldExpression class="java.lang.String"><![CDATA["Invoice # " + $F{INVOICE_NO} + " total"]></textFieldExpression>
    </textField>
  </band>
</detail>
```

Even though you passed the parameter CURRENT\_CUST to the subreport, you must tell the report what type the parameter is, using the class= attribute of the <parameter> tag. Next comes the query string (JasperReports is very particular about the order of the tags within the file). Here you reference the value of CURRENT\_CUST as \$P{CURRENT\_CUST}. The field names refer to column names in the ORDERS table, which you can then reference inside the <textFieldExpression> tag.

These are the only changes you need to make to add a subreport; you do not need to change any code in the EGL report driver program. If you wish to include other sources of data or complex calculations, however, you will need to create a report handler (see *Creating an EGL report handler*).

You can create nested subreports. For example, you might call up line items for each invoice from a third table. This would involve adding the information within a <subreport> tag to the subreport file my\_subreport.jasper, and creating a separate design file for the nested subreport (you might call it my\_invoice\_subreport.jasper). There is no limit to the depth to which you can nest subreports.

#### Related Tasks

“Migrating EGL code to EGL V6.0 iFix” on page 103

“Creating the report design file” on page 254

#### Related Reference

“EGL report handler”

---

## EGL report handler

The standard features of the JasperReports open-source reporting library allow you to create quite complex output. If you need more sophisticated reports, you may create an EGL report handler.

The EGL report handler is an EGL handler part of type JasperReport. The JasperReports engine sees the report handler as a scriptletClass. For those unfamiliar with Java, the implication is that the JasperReports design file can call methods (similar to functions) of the report handler.

Here is a quick list of common uses for a report handler, with details to follow afterward:

- Responding to specific, predefined events while JasperReports fills the report
- Providing functions that you explicitly invoke from the XML design file
- Storing data to be retrieved for subreports

You can create the report handler from a blank file or use the EGL New Report Handler wizard. The report handler wizard starts you off with an outline that includes the name of each predefined (event-related) function that the JasperReports engine can invoke while filling the report. For a complete list of these events, see *Predefined EGL report handler functions*.

## Responding to events

EGL has a number of predefined function names that correspond to events that may occur when JasperReports fills a report. Such events include entering or leaving a page, beginning or ending a line item, and others. When one of these events occurs, the JasperReports engine checks to see if the report handler contains a function that corresponds to the event. If so, the JasperReports engine will invoke that function automatically.

One of the events that EGL recognizes is the initialization of a user-defined group. The `<group>` tag in the XML design source file allows you to specify an expression that defines a group. For example, you might group customers together whose numbers range from 2000 to 2999. You can print subtotals for the group through the standard XML design source tags, or you can perform more complex manipulations using the report handler.

## Creating explicitly invoked functions

You cannot perform any complex (multi-line) Java coding within the JasperReports XML source file itself. By writing EGL code in the report handler, however, you can create functions that the JasperReports engine will be able to access at run time. These functions do not have to be tied to any specific event or be invoked from within one of the predefined event-related functions. For example, you might create a function that looks up a SQL variable of type DATE and returns the date in a formatted character string. You can call this function directly from the XML source that prints the detail for a transaction.

The operation works because EGL generates a scriptletClass from the report handler source file. The scriptletClass is a Java class that the JasperReports engine can access, and during printing, the engine can invoke the methods of that class. Those methods are the functions you created.

The report handler has full access to other EGL resources, such as record parts, system functions, and libraries.

## Storing and retrieving data

Passing data to a subprogram is a two-step process.

1. Use the system function `ReportLib.addReportData()` in the report handler to store data for later use by a subreport.
2. Inside the design file's `<subreport>` tag, set up a `<dataSourceExpression>` tag to pass data to the subreport's design file. Within the `<dataSourceExpression>` tag, invoke the built-in `getDataSource()` method of the report handler. This method

returns the data in an internal format. You must cast the return value as a `JRDataSource` for the subreport to be able to use the data.

The subreport can then use the retrieved data to provide detail for each line item.

For more details and examples, see *Creating an EGL report handler* and *Additional EGL report handler functions*.

## Generated output

When generating a report handler, EGL creates these files:

- `handlerName.java` is the actual report handler. JasperReports makes calls to this program using Java data types.
- `handlerName_lib.java` is a library that contains translated EGL functions. The report handler (`handlerName.java`) calls these functions with data types translated from Java.

*handlerName*

Name of the EGL report handler source file (less .egl extension)

At generation time, EGL uses the EGL report handler source file to create a `JRDefaultScriptlet` class, which is a subclass of the JasperReports scriptlet class. That subclass contains a method for each function that you coded. At run time, the JasperReports engine checks for a `scriptletClass` attribute in the `<jasperReports>` tag. If this attribute exists, the report engine loads the scriptlet class and makes the class methods available to the report design. For more information on JasperReports scriptlets and scriptlet class, see the JasperReports documentation.

When EGL generates .java files, the class names are lower case. Be sure that any class name you enter in an XML design document is in lower case.

### Related concepts

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

"Event handling in EGL" on page 99

### Related tasks

"Migrating EGL code to EGL V6.0 iFix" on page 103

"Creating subreports" on page 263

"Writing code to drive a report" on page 257

### Related reference

"Additional EGL report handler functions" on page 272

"EGL library ReportLib" on page 990

"Predefined EGL report handler functions" on page 271

---

## Creating an EGL report handler

An EGL report handler provides blocks of code that the JasperReports engine can access at run time. Predefined function names tie some of these code blocks to events that occur when JasperReports fills a report. Such events might include the beginning or end of a page, of a line item, or of the report itself. You can call other, custom functions directly from the XML design file source.

To create an EGL report handler, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one.
2. In the workbench, click **File > New > Other**.
3. In the New window, expand **EGL**.
4. Click **Report Handler**.
5. Click **Next**.
6. Select the project or folder that will contain the EGL source file, then select a package.
7. In the EGL Source File Name field, type the name report handler source file. As the report handler name will be identical to the file name, choose a file name that adheres to EGL part name conventions (for example, myReportHandler).
8. Click **Finish**.

The New EGL Report Handler wizard will give you a list of function names that correspond to report fill events. JasperReports will invoke "beforePageInit()", for example, before entering a page. It is up to you to create the code for these functions.

Alternatively, you can add this template information to an existing file by following these steps:

1. Create a new EGL source file.
2. Type **handler** followed by **Ctrl+space**.

The remainder of this topic contains code examples that show the following:

- The outline of a generic report handler
- How to get report parameters in a report-handler
- How to get and set report variables
- How to get field values
- How to add a report data record
- How to pass report data to an XML design document
- How to invoke a custom report handler function from the XML design document

These examples barely begin to address the complexities possible in a report handler. For more detail, see the JasperReports documentation.

## Report handler template

This is the template code created by the New EGL Report Handler wizard:

```
handler handlerName type jasperReport

// Use Declarations
use usePartReference;

// Constant Declarations)
const constantName constantType = literal;

// Data Declarations
identifierName declarationType;

// Pre-defined Jasper callback functions
function beforeReportInit()
end

function afterReportInit()
```

```

end

function beforePageInit()
end

function afterPageInit()
end

function beforeColumnInit()
end

function afterColumnInit()
end

function beforeGroupInit(stringVariable string)
end

function afterGroupInit(stringVariable string)
end

function beforeDetailEval()
end

function afterDetailEval()
end
end

```

## Getting report parameters

Reports can contain three types of items: parameters (set in the XML file and not changed), variables (changeable by the XML file or the report handler) and fields (keyed to names in the data source). The following code snippet shows how to get report parameters in a report handler:

```

handler my_report_handler type jasperReport

// Data Declarations
report_title String;

// Jasper callback function
function beforeReportInit()
  report_title = getReportParameter("ReportTitle");
end

end

```

## Getting and setting report variables

The following code snippet shows how to get and set report variables in a report handler:

```

handler my_report_handler type jasperReport

// Data Declarations
item_count int;

// Jasper callback function
function beforeDetailEval()
  item_count = getReportVariableValue("itemCount");
end

function afterDetailEval()
  setReportVariableValue("itemCount", (item_count + 1));
end
end

```

Remember to match variable types in the report handler with those in your XML source file.

## Getting report field values

The following example code snippet shows how to get report field values in a report handler:

```
handler my_report_handler type jasperReport

// Data Declarations
employee_first_name String;

// Jasper callback function
function beforeColumnInit()
    employee_first_name = getFieldValue("fName");
end
end
```

## Saving report data in the report handler

The following example code shows how to save a customer record under the name "saveCustomer" for later access:

```
handler my_report_handler type jasperReport

// Data Declarations
customer_array customerRecordType[];
c                customerRecordType;

// Jasper callback function
function beforeReportInit()
    customer ReportData;

    //create the ReportData object for the Customer subreport
    c.customer_num = getFieldValue("c_customer_num");
    c.fname        = getFieldValue("c_fname");
    c.lname        = getFieldValue("c_lname");
    c.company      = getFieldValue("c_company");
    c.address1     = getFieldValue("c_address1");
    c.address2     = getFieldValue("c_address2");
    c.city         = getFieldValue("c_city");
    c.state        = getFieldValue("c_state");
    c.zipcode      = getFieldValue("c_zipcode");
    c.phone        = getFieldValue("c_phone");
    customer_array.appendElement(c);
    customer.data = customer_array;
    addReportData(customer, "saveCustomer");
end
end
```

## Retrieving report data in the XML file

Now that we have saved the report data in the report handler, we can retrieve it in the XML source file and pass it to a subreport:

```
<jasperReport name="MasterReport" ... scriptletClass="subreports.my_report_handler">
...

<subreport>
<dataSourceExpression>
    <![CDATA[(JRDataSource)((subreports.SubReportHandler)
        ${REPORT_SCRIPTLET}).getReportData( new String("saveCustomer"))]]>;
</dataSourceExpression>
<subreportExpression class="java.lang.String">
    <![CDATA["C:/RAD/workspaces/customer_subreport.jasper"]]]>;
</subreportExpression>
```



```

</subreport>

...
</jasperReport>

```

## Invoking a function from the XML design document

Here's a report handler with one very simple function:

```

handler my_report_handler type jasperReport

function hello () returns (String)
    return("Hello, world!");
end
end

```

Invoke it from the XML design document with this code:

```

<jasperReport name="MasterReport" ... scriptletClass="my_package.my_report_handler">
...

<summary>
<band height="40">
<textField>
<reportElement positionType="Float" x="0" y="20" width="500" height="15"/>
<textElement textAlignment="Center">
<font reportFont="Arial_Bold" size="10"/>
</textElement>
<textFieldExpression class="java.lang.String">
<![CDATA[ ((my_package.my_report_handler)$P{REPORT_SCRIPTLET}).hello() ]]>
</textFieldExpression>
</textField>
</band>
</summary>

...
</jasperReport>

```

The phrase "Hello, world!" will print at the end of the report.

### Related concepts

"EGL report creation process overview" on page 252

"EGL reports overview" on page 251

### Related tasks

"Creating an EGL source file" on page 130

"Using report templates" on page 259

### Related Reference

"Additional EGL report handler functions" on page 272

"EGL library ReportLib" on page 990

"EGL report handler" on page 264

"Predefined EGL report handler functions" on page 271

---

## Predefined EGL report handler functions

The Report Handler provides the following predefined function names, corresponding to events during report fill. Add your code to these functions to create additional detail for your reports:

Function	Where the function operates
<code>beforeReportInit();</code>	Before report initialization
<code>afterReportInit();</code>	After report initialization
<code>beforePageInit();</code>	Entering a page
<code>afterPageInit();</code>	Leaving a page
<code>beforeColumnInit();</code>	Before column initialization
<code>afterColumnInit();</code>	After column initialization
<code>beforeGroupInit (groupName String);</code>	Before group initialization, where <i>groupName</i> refers to a <code>&lt;group&gt;</code> tag in the design document
<code>afterGroupInit(groupName String);</code>	After group initialization
<code>beforeDetailEval();</code>	Before each row is printed
<code>afterDetailEval();</code>	After each row is printed

Within one of these functions, you can make calls to other functions. For example, you can make a call to `setReportVariable()`, as follows:

```
function afterGroupInit(groupName String)
  if (groupName == "cat")
    setReportVariableValue ("NewGroupName", "dog");
  else
    setReportVariableValue ("NewGroupName", groupName);
  end
end
```

You can also create your own functions. See *JasperReports* documentation for information about creating custom functions.

For examples using predefined report handler functions, see *Creating an EGL report handler*.

### Related concepts

“EGL report creation process overview” on page 252

“EGL reports overview” on page 251

### Related tasks

“Migrating EGL code to EGL V6.0 iFix” on page 103

### Related reference

“Additional EGL report handler functions” on page 272

“EGL library ReportLib” on page 990

“EGL report handler” on page 264

“Report and ReportData parts” on page 262

---

## Additional EGL report handler functions

EGL has several ReportLib functions that increase the power of the report handler. The JasperReport design file can invoke an additional predefined Java method, `getDataSource()`.

JasperReports distinguishes between three types of entities that the report engine evaluates at run time:

- parameters (`$P{parameter_name}`) keep the same value throughout the process. You can use them to pass a value from the report engine to the report handler (but not the other way around).
- variables (`$V{variable_name}`) can be read or set by the report handler. The design file can change their values as well.
- fields (`$F{field_name}`) map data from the data source to the printed report. The report handler can read values from fields but cannot change those values.

You can invoke any of the ReportLib functions in the following sections from within the report handler. Invoke `getDataSource()`, which is not a ReportLib function, from the XML design file source.

### Function for getting report parameters

Function	Purpose
<b>getReportParameter</b> ( <i>parameter</i> <b>String</b> <u>in</u> )	Returns the value of the specified parameter from the report that is being filled. The returned value is of type ANY.

### Functions for setting and getting report variables

These functions give you access at run time to variables that the report uses.

Function	Purpose
<b>getReportVariableValue</b> ( <i>variable</i> <b>String</b> <u>in</u> )	Returns the value of the specified variable from the report that is being filled. The returned value is of type ANY.
<b>setReportVariableValue</b> ( <i>variable</i> <b>String</b> <u>in</u> , <i>value</i> <b>Any</b> <u>in</u> )	Assigns the value to the specified variable.

### Function for getting field values

Function	Purpose
<b>getFieldValue</b> ( <i>fieldName</i> <b>String</b> <u>in</u> )	Returns the value of the specified field value for the row currently being processed. The returned value is of type ANY.

### Functions for storing or retrieving data for subreports

A subreport is a report that you call from within another report. To pass report data to a subreport, the report handler first saves the information using `addReportData()`. The JasperReports engine can retrieve this data by means of `getDataSource()`, which is described later in this topic. To retrieve previously saved data within the report handler, use the equivalent function `getReportData()`.

Function	Purpose
<b>addReportData</b> ( <i>rd</i> ReportData <u>in</u> , <i>dataID</i> String <u>in</u> )	Associates data (as stored in a variable of type ReportData) with the name <i>dataID</i> .
<b>getReportData</b> ( <i>dataID</i> String <u>in</u> )	Retrieves data that you previously stored under <i>dataID</i> using addReportData(). getReportData() returns a value of type ReportData.

## Java methods available to the report design file

To access data that you will pass to a subreport, call the `getDataSource()` method of the EGL report library from your XML report design file.

Function	Purpose
<b>getDataSource</b> ( <i>dataID</i> String <u>in</u> )	Within a report design file, retrieves data that you previously stored under <i>dataID</i> when you invoked addReportData() in the report handler.  getDataSource() returns the data in an internal format. You must cast the return value as a JRDataSource for the subreport to be able to use that data.

For examples how to use the functions described in this topic, see *Creating an EGL report handler*.

### Related concepts

"EGL report creation process overview" on page 252

"EGL reports overview" on page 251

### Related tasks

"Migrating EGL code to EGL V6.0 iFix" on page 103

### Related reference

"EGL library ReportLib" on page 990

"EGL report handler" on page 264

"Predefined EGL report handler functions" on page 271

"Report and ReportData parts" on page 262

---

## Generating files for and running a report

You must have the following files in the specified locations:

- You will need a compiled JasperReports design file (extension *jasper*) somewhere on your system (you specify the location in your report driver). For more information, see *Creating the report design file*.
- Your EGL report driver source file (extension *egl*) must be in the package directory under EGLSource.
- Make sure you have a build descriptor file, probably in the EGLSource directory. See *Generation in the workbench*.
- If you want a report handler, that source file (extension *egl*) must also be in the package directory under EGLSource.

- If your report driver accesses a database, you might need to tell EGL where to find the appropriate Java Database Connectivity (JDBC) driver. You might need to add an external Java Archive (JAR) file to your project properties.

To create and fill a report for an EGL project, follow these steps:

1. Build the EGL project by selecting **Project > Build All** or **Project > Clean**. EGL automatically generates Java code from any EGL source files that have changed since the last build and compiles any changed design document source files.
2. Run the Java program that has the report invocation code. One way to do this in the Package Explorer is to navigate to and right-click the .java file that contains the code. Then select **Run > Java Application** from the popup menu.

When you create a report, the JasperReports engine first creates an intermediate destination file (extension *jrprint*), from which it can create multiple export files in different formats (.pdf, .html, .xml, .txt, or .csv). You specify locations for all of these files in the report driver.

**Related concepts** “EGL report creation process overview” on page 252

“EGL reports overview” on page 251

“Generation in the workbench” on page 416

#### Related tasks

“Migrating EGL code to EGL V6.0 iFix” on page 103

“Creating the report design file” on page 254

“Exporting reports”

“Writing code to drive a report” on page 257

#### Related reference

“EGL library ReportLib” on page 990

“EGL report handler” on page 264

---

## Exporting reports

JasperReports saves filled report data as an intermediate destination file (extension *jrprint*), from which it can create multiple export files. You can export filled reports as PDF, HTML, XML, CSV (comma-separated values, readable by Excel), and plain text output. Use the **reportLib.exportReport()** function in the EGL report driver, paired with a statement that sets an export file location (see the example that follows) to export a version of a report.

The **reportLib.exportReport()** function recognizes the following parameters:

- ExportFormat.html
- ExportFormat.pdf
- ExportFormat.text
- ExportFormat.xml
- ExportFormat.csv

For example, the following code causes JasperReports to export a report as a PDF file:

```
myReport.ReportExportFile = "c:\\temp\\my_report.pdf";
reportLib.exportReport(myReport, ExportFormat.pdf);
```

**Important:** JasperReports does not automatically refresh exported reports. If you change the report design or if data changes, you must refill and re-export the report.

**Related concepts**

“EGL report creation process overview” on page 252

“EGL reports overview” on page 251

**Related tasks**

“Generating files for and running a report” on page 273

“Writing code to drive a report” on page 257

**Related reference**

“EGL library ReportLib” on page 990

“EGL report handler” on page 264



---

## Working with files and databases

---

### SQL support

As shown in the next table, EGL-generated code can access a relational database on any of the target systems.

Target System	Support for access of relational databases
AIX, HP-UX, iSeries, Linux, Solaris, UNIX System Services, Windows 2000/NT/XP	JDBC provides access to DB2 UDB, Oracle, Informix, or Microsoft SQL Server

As you work on a program, you can code SQL statements as you would when coding programs in most other languages. To ease your way, EGL provides SQL statement templates for you to fill.

Alternatively, you can use an SQL record as the I/O object when you code an EGL statement. Using the record in this way means that you access a database either by customizing an SQL statement provided to you or by relying on a default that removes the need to code SQL.

In either case, be aware of these aspects of EGL support for SQL:

- If you want to test for a null in a particular table column, you must receive the column value into an SQL record, into a record item that is nullable. For details, see *Testing for and setting NULL* described later.
- In the overview sections that follow (and in keeping with SQL terminology), each item that is referenced in an SQL statement is called a *host variable*. The word *host* refers to the language that embeds the SQL statement; in this case, to the EGL procedural language. A host variable in an SQL statement is preceded by a colon, as in this example:

```
select empnum, empname
from employee
where empnum >= :myRecord.empnum
for update of empname
```

### EGL statements and SQL

The next table lists the EGL keywords that you can use to access a relational database. Included in this table is an outline of the SQL statements that correspond to each keyword. When you code an EGL **add** statement, for example, you generate an SQL INSERT statement.

In many business applications, you use the EGL **open** statement and various kinds of **get by position** statements. The code helps you to declare, open, and process a *cursor*, which is a runtime entity that acts as follows:

- Returns a *result set* with a list of the rows that fulfill your search criteria
- Points to a specific row in the result set

You can use the EGL **open** statement to call a stored procedure. That procedure is composed of logic that is written outside of EGL, is stored in the database management system, and also returns a result set. (In addition, you can use the EGL **execute** statement to call a stored procedure.)



Later sections give details on processing a result set.

If you intend to code SQL statements explicitly, you use the EGL **execute** statement and possibly the EGL **prepare** statement.

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
<b>add</b>  Places a row in a database; or (if you use a dynamic array of SQL records), places a set of rows based on the content of successive elements of the array.	INSERT row (as occurs repeatedly, if you specify a dynamic array).	Yes
<b>close</b>  Releases unprocessed rows.	CLOSE cursor.	No
<b>delete</b>  Deletes a row from a database.	DELETE row. The row was selected in either of two ways: <ul style="list-style-type: none"> <li>• When you invoked a <b>get</b> statement with the <b>forUpdate</b> option (as appropriate when you wish to select the first of several rows that have the same key value)</li> <li>• When you invoked an <b>open</b> statement with the <b>forUpdate</b> option and then a <b>get next</b> statement (as appropriate when you wish to select a set of rows and to process the retrieved data in a loop)</li> </ul>	No
<b>forEach</b>  Marks the start of a set of statements that run in a loop. The first iteration occurs only if a specified result set is available and continues (in most cases) until the last row in that result set is processed.	EGL converts a <b>forEach</b> statement into an SQL FETCH statement that runs inside a loop.	No
<b>freeSQL</b>  Frees any resources associated with a dynamically prepared SQL statement, closing any open cursor associated with that SQL statement.		No

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
<p><b>get</b> (also called <b>get by key value</b>)</p> <p>Reads a single row from a database; or (if you use a dynamic array of SQL records), reads successive rows into successive elements in the array.</p>	<p>SELECT row, but only if you set the option <code>singleRow</code>. Otherwise, the following rules apply:</p> <ul style="list-style-type: none"> <li>EGL converts a <b>get</b> statement to this: <ul style="list-style-type: none"> <li>– DECLARE cursor with SELECT or (if you set the <code>forUpdate</code> option) with SELECT FOR UPDATE.</li> <li>– OPEN cursor.</li> <li>– FETCH row.</li> </ul> </li> <li>If you did not specify the option <code>forUpdate</code>, EGL also closes the cursor.</li> <li>The <code>singleRow</code> and <code>forUpdate</code> options are not supported with dynamic arrays; in that case, the EGL runtime declares and opens a cursor, fetches a series of rows, and closes the cursor.</li> </ul>	Yes
<p><b>get absolute</b></p> <p>Reads a numerically specified row in a result set that was selected by an <b>open</b> statement.</p>	EGL converts a <b>get absolute</b> statement to an SQL FETCH statement.	No
<p><b>get current</b></p> <p>Reads the row at which the cursor is already positioned in a result set that was selected by an <b>open</b> statement.</p>	EGL converts a <b>get current</b> statement to an SQL FETCH statement.	No
<p><b>get first</b></p> <p>Reads the first row in a result set that was selected by an <b>open</b> statement.</p>	EGL converts a <b>get first</b> statement to an SQL FETCH statement.	No
<p><b>get last</b></p> <p>Reads the last row in a result set that was selected by an <b>open</b> statement.</p>	EGL converts a <b>get last</b> statement to an SQL FETCH statement.	No
<p><b>get next</b></p> <p>Reads the next row in a result set that was selected by an <b>open</b> statement.</p>	EGL converts a <b>get next</b> statement to an SQL FETCH statement.	No
<p><b>get previous</b></p> <p>Reads the previous row in a result set that was selected by an <b>open</b> statement.</p>	EGL converts a <b>get previous</b> statement to an SQL FETCH statement.	No

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
<p>get relative</p> <p>Reads a numerically specified row in a result set that was selected by an <b>open</b> statement. The row is identified in relation to the cursor position in the result set.</p>	<p>EGL converts a <b>get relative</b> statement to an SQL FETCH statement.</p>	No
<p>execute</p> <p>Lets you run an SQL data-definition statement (of type CREATE TABLE, for example); or a data-manipulation statement (of type INSERT or UPDATE, for example); or a prepared SQL statement that does not begin with a SELECT clause.</p>	<p>The SQL statement you write is made available to the database management system.</p> <p>The primary use of <b>execute</b> is to code a single SQL statement that is fully formatted at generation time, as in this example--</p> <pre> try   execute     #sql{      // no space after "#sql"       delete       from EMPLOYEE       where department =         :myRecord.department     }; onException   myErrorHandler(10); end </pre> <p>A fully formatted SQL statement may include host variables in the WHERE clause.</p>	Yes
<p>open</p> <p>Selects a set of rows from a relational database for later retrieval with <b>get next</b> statements.</p>	<p>EGL converts an <b>open</b> statement to a CALL statement (for accessing a stored procedure) or to these statements:</p> <ul style="list-style-type: none"> <li>• DECLARE cursor with SELECT or with SELECT FOR UPDATE.</li> <li>• OPEN cursor.</li> </ul>	Yes

Keyword/Purpose	Outline of SQL statements	Can you modify the SQL?
<p><b>prepare</b></p> <p>Specifies an SQL PREPARE statement, which optionally includes details that are known only at run time; you run the prepared SQL statement by running an EGL execute statement or (if the SQL statement begins with SELECT) by running an EGL open or get statement.</p>	<p>EGL converts a <b>prepare</b> statement to an SQL PREPARE statement, which is always constructed at run time. In the following example of an EGL <b>prepare</b> statement, each parameter marker (?) is resolved by the USING clause in the subsequent <b>execute</b> statement:</p> <pre> myString =   "insert into myTable " +   "(empnum, empname) " +   "value ?, ?";  try   prepare myStatement     from myString; onException   // exit the program   myErrorHandler(12); end  try   execute myStatement     using :myRecord.empnum,           :myRecord.empname; onException   myErrorHandler(15); end </pre>	Yes
<p><b>replace</b></p> <p>Puts a changed row back into a database.</p>	<p>UPDATE row. The row was selected in either of two ways:</p> <ul style="list-style-type: none"> <li>• When you invoked a <b>get</b> statement with the forUpdate option (as appropriate when you wish to select the first of several rows that have the same key value); or</li> <li>• When you invoked an <b>open</b> statement with the forUpdate option and then a <b>get next</b> statement (as appropriate when you wish to select a set of rows and to process the retrieved data in a loop).</li> </ul>	Yes

**Note:** Under no circumstances can you update multiple database tables by coding a single EGL statement.

## Result-set processing

A common way to update a series of rows is as follows:

1. Declare and open a cursor by running an EGL **open** statement with the option forUpdate; that option causes the selected rows to be locked for subsequent update or deletion
2. Fetch a row by running an EGL **get next** statement
3. Do the following in a loop:
  - a. Change the data in the host variables into which you retrieved data
  - b. Update the row by running an EGL **replace** statement
  - c. Fetch another row by running an EGL **get next** statement
4. Commit changes by running the EGL function **commit**.

The statements that open the cursor and that act on the rows of that cursor are related to each other by a result-set identifier, which must be unique across all result-set identifiers, program variables, and program parameters within the program. You specify that identifier in the **open** statement that opens the cursor, and you reference the same identifier in the **get next**, **delete**, and **replace** statements that affect an individual row, as well as on the **close** statement that closes the cursor. For additional details, see *resultSetID*.

The following code shows how to update a series of rows when you are coding the SQL yourself:

```

VGVar.handleHardIOErrors = 1;

try
  open selectEmp forUpdate with
  #sql{          // no space after "#sql"
    select empname
    from EMPLOYEE
    where empnum >= :myRecord.empnum
    for update of empname
  };

onException
  myErrorHandler(8);    // exits program
end

try
  get next from selectEmp into :myRecord.empname;
onException
  if (sysVar.sqlcode != 100)
    myErrorHandler(8); // exit the program
  end
end

while (sysVar.sqlcode != 100)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    execute
    #sql{
      update EMPLOYEE
      set empname = :empname
      where current of selectEmp
    };
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next from selectEmp into :myRecord.empname;
  onException
    if (sysVar.sqlcode != 100)
      myErrorHandler(8); // exits program
    end
  end

end // end while; cursor is closed automatically
// when the last row in the result set is read

sysLib.commit;

```

If you wish to avoid some of the complexity in the previous example, consider SQL records. Their use allows you to streamline your code and to use I/O error values that do not vary across database management systems. The next example is equivalent to the previous one but uses an SQL record called emp:

```

VGVar.handleHardIOErrors = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(8); // exits program
end

try
  get next emp;
onException
  if (sysVar.sqlcode not noRecordFound)
    myErrorHandler(8); // exit the program
  end
end

while (sysVar.sqlcode not noRecordFound)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    replace emp;
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next emp;
  on exception
    if (sysVar.sqlcode not noRecordFound)
      myErrorHandler(8); // exits program
    end
  end
end // end while; cursor is closed automatically
// when the last row in the result set is read

sysLib.commit;

```

Later sections describe SQL records.

## SQL records and their uses

An SQL record is a variable that is based on an SQL record part. This type of record allows you to interact with a relational database as though you were accessing a file. If the variable EMP is based on an SQL record part that references the database table EMPLOYEE, for example, you can use EMP in an EGL **add** statement:

```
add EMP;
```

In this case, EGL inserts the data from EMP into EMPLOYEE. The SQL record also includes state information so that after the EGL statement runs, you can test the SQL record to perform tasks conditionally, in accordance with the I/O error value that resulted from database access:

```

VGVar.handleHardIOErrors = 1;

try
  add EMP;
onException
  if (EMP is unique) // if a table row
                    // had the same key
    myErrorHandler(8);
  end
end

```

An SQL record like EMP allows you to interact with a relational database as follows:

- Declare an SQL record part and the related SQL record
- Define a set of EGL statements that each use the SQL record as an I/O object
- Accept the default behavior of the EGL statements or make the SQL changes that are appropriate for your business logic

### Declaring an SQL record part and the related record

You declare an SQL record part and associate each of the record items with a column in a relational table or view. You can let EGL make this association automatically by way of the EGL editor's retrieve feature, as described later in *Database access at declaration time*.

If the SQL record part is not a fixed record part, you can include primitive fields as well as other variables. You are especially likely to include the following kinds of variables:

- Other SQL records. The presence of each represents a one-to-one relationship between the parent and child tables.
- Arrays of SQL records. The presence of each represents a one-to-many relationship between the parent and child tables.

Only fields of a primitive type can represent a database column.

If level numbers precede the fields, the SQL record part is a fixed record part. The following rules apply:

- The structure in each SQL record part must be *flat* (without hierarchy)
- All of the fields must be primitive fields, but not of type BLOB, CLOB, or STRING
- None of the record fields can be a structure-field array

After you declare an SQL record part, you declare an SQL record that is based on that part.

### Defining the SQL-related EGL statements

You can define a set of EGL statements that each use the SQL record as the I/O object in the statement. For each statement, EGL provides an *implicit SQL statement*, which is not in the source but is implied by the combination of SQL record and EGL statement. In the case of an EGL **add** statement, for example, an implicit SQL INSERT statement places the value of a given record item into the associated table column. If your SQL record includes a record item for which no table column was assigned, EGL forms the implicit SQL statement on the assumption that the name of the record item is identical to the name of the column.

**Using implicit SELECT statements:** When you define an EGL statement that uses an SQL record and that generates either an SQL SELECT statement or a cursor declaration, EGL provides an implicit SQL SELECT statement. (That statement is embedded in the cursor declaration, if any.) For example, you might declare a variable that is named EMP and is based on the following record part:

```
Record Employee type sqlRecord
{ tableNames = ["EMPLOYEE"],
  keyItems = ["empnum"] }
empnum decimal(6,0);
empname char(40);
end
```

Then, you might code a **get** statement:

```
get EMP;
```

The implicit SQL SELECT statement is as follows:

```
SELECT empnum, empname  
FROM   EMPLOYEE  
WHERE  empnum = :empnum
```

EGL also places an INTO clause into the standalone SELECT statement (if no cursor declaration is involved) or into the FETCH statement associated with the cursor. The INTO clause lists the host variables that receive values from the columns listed in the first clause of the SELECT statement:

```
INTO :empnum, :empname
```

The implicit SELECT statement reads each column value into the corresponding host variable; references the tables specified in the SQL record; and has a search criterion (a WHERE clause) that depends on *a combination of two factors*:

- The value you specified for the record property **defaultSelectCondition**; and
- A relationship (such as an equality) between two sets of values:
  - Names of the columns that constitute the table key
  - Values of the host variables that constitute the record key

A special situation is in effect if you read data into a dynamic array of SQL records, as is possible with the **get** statement:

- A cursor is open, successive rows from the database are read into successive elements of the array, the result set is freed, and the cursor is closed.
- If you do not specify an SQL statement, the search criterion depends on the record property **defaultSelectCondition**, but also depends on a relationship (specifically, a greater-than-or-equal-to relationship) between the following sets of values:
  - Names of columns, as specified indirectly when you specify items in the EGL statement
  - Values of those items

Any host variables specified in the property **defaultSelectCondition** must be outside the SQL record that is the basis of the dynamic array.

For details on the implicit SELECT statement, which vary by keyword, see *get* and *open*.

**Using SQL records with cursors:** When you are using SQL records, you can relate cursor-processing statements by using the same SQL record in several EGL statements, as you can by using a result-set identifier. However, any cross-statement relationship that is indicated by a result-set identifier takes precedence over a relationship indicated by the SQL record; and in some cases you must specify a resultSetID.

In addition, only one cursor can be open for a particular SQL record. If an EGL statement opens a cursor when another cursor is open for the same SQL record, the generated code automatically closes the first cursor.

## Customizing the SQL statements

Given an EGL statement that uses an SQL record as the I/O object, you can progress in either of two ways:



- You can accept the implicit SQL statement. In this case, changes made to the SQL record part affect the SQL statements used at run time. If you later indicate that a different record item is to be used as the key of the SQL record, for example, EGL changes the implicit SELECT statement used in any cursor declaration that is based on that SQL record part.
- You can choose instead to make the SQL statement explicit. In this case, the details of that SQL statement are isolated from the SQL record part, and any subsequent changes made to the SQL record part have no effect on the SQL statement that is used at run time.

If you remove an explicit SQL statement from the source, the implicit SQL statement (if any) is again available at generation time.

### Example of using a record in a record

To allow a program to retrieve data for a series of employees in a department, you can create two record parts and a function, as follows:

```
DataItem DeptNo { column = deptNo } end

Record Dept type SQLRecord
  deptNo DeptNo;
  managerID CHAR(6);
  employees Employee[];
end

Record Employee type SQLRecord
  employeeID CHAR(6);
  empDeptNo DeptNo;
end

Function getDeptEmployees(dept Dept)
  get dept.employees usingKeys dept.deptNo;
end
```

### Testing for and setting NULL

The build descriptor option **itemsNullable** controls the circumstance in which EGL internally maintains a null indicator for a primitive variable in your code. If you accept the default setting for that option, EGL internally maintains a null indicator only for each variable that has the following characteristics:

- Is in an SQL record
- Is declared with the property **isNullable** set to *yes*

Do not code host variables for null indicators in your SQL statements, as you might in some languages. To test for NULL in a nullable host variable, use an EGL **if** statement. You also can test for retrieval of a truncated value, but only when a null indicator is available.

You can null an SQL table column in either of two ways:

- Use an EGL **set** statement to null a nullable host variable, then write the related SQL record to the database; or
- Use the appropriate SQL syntax, either by writing an SQL statement from scratch or by customizing an SQL statement that is associated with the EGL **add** or **replace** statement.

For additional details on null processing, see *isNullable* and *itemsNullable*.

## Database access at declaration time

You receive the following benefits from accessing (at declaration time) a database that has similar characteristics to the database that your code will access at run time:

- If you access a database table or view that is equivalent to a table or view associated with an SQL record, you can use the retrieve feature of the EGL part editor to create or overwrite the record items. The retrieve feature accesses information stored in the database management system so that the number and data characteristics of the created items reflect the number and characteristics of the table columns. After you invoke the retrieve feature, you can rename record items, delete record items, and make other changes to the SQL record.
- Your access of an appropriately structured database at declaration time helps to ensure that your SQL statements will be valid in relation to an equivalent database at run time.

The retrieve feature creates record items that each have the same name (or almost the same name) as the related table column.

You cannot retrieve a view that is defined with the DB2 condition WITH CHECK OPTIONS.

For further details on using the retrieve feature, see *Retrieving SQL table data*. For details on naming, see *Setting preferences for SQL retrieve*.

To access a database at declaration time, specify connection information in a preferences page, as described in *Setting preferences for SQL database connections*.

### Related concepts

"Dynamic SQL" on page 288  
"Logical unit of work" on page 395  
"resultSetID" on page 867

### Related tasks

"Retrieving SQL table data" on page 299  
"Setting preferences for SQL database connections" on page 121  
"Setting preferences for SQL retrieve" on page 123

### Related reference

"add" on page 661  
"close" on page 669  
"Database authorization and table names" on page 557  
"Default database" on page 298  
"delete" on page 673  
"execute" on page 677  
"get" on page 687  
"get next" on page 701  
"Informix and EGL" on page 299  
"isNullable" on page 813  
"itemsNullable" on page 482  
"open" on page 722  
"prepare" on page 736  
"replace" on page 738  
"SQL data codes and EGL host variables" on page 874

"SQL examples"

"SQL item properties" on page 68

"SQL record internals" on page 876

"SQL record part in EGL source format" on page 877

"Testing for and setting NULL" on page 286

## Dynamic SQL

The SQL statement associated with an EGL statement can be specified statically, with every detail in place at generation time. When dynamic SQL is in effect, however, the SQL statement is built at run time, each time that the EGL statement is invoked.

Use of dynamic SQL decreases the speed of runtime processing, but lets you vary a database operation in response to a runtime value:

- For a database query, you may want to vary the selection criteria, how data is aggregated, or the order in which rows are returned; those details are controlled by the WHERE, HAVING, GROUP BY, and ORDER BY clauses. In this case, you can use the prepare statement.
- For many kinds of operations, you may want a runtime value to determine which table to access. You can accomplish dynamic specification of a table in either of two ways:
  - Use the prepare statement; or
  - Use an SQL record and specify a value for the property **tableNameVariables**, as described in *SQL record part in EGL source format*.

### Related concepts

"SQL support" on page 277

### Related reference

"Database authorization and table names" on page 557

"prepare" on page 736

"SQL record part in EGL source format" on page 877

## SQL examples

You can access an SQL data base in any of these ways:

- By hand-coding an SQL statement whose format is known at generation time.
- By using an SQL record as the I/O object of an EGL statement, when the format of the SQL statement is known at generation time--
  - If you place an explicit SQL statement in the EGL source, that SQL statement is used at run time;
  - Otherwise, an implicit SQL statement is used at run time.
- By coding an EGL **prepare** statement, which generates an SQL PREPARE statement that in turn creates an SQL statement at run time.

In every case, you can use an SQL record as a memory area and to provide a simple way to test for successful operation. The examples in this section assume that a record part is declared in an EGL source file and that a record based on the part was declared in a program in that file:

- The SQL record part is as follows--

```
Record Employee type sqlRecord
{
    tableName = ["employee"],
    keyItems = ["empnum"],
```

```

        defaultSelectCondition =
        #sqlCondition{    // no space
                        // between #sqlCondition
                        // and the brace
            aTableColumn = 4    -- start each SQL comment
                                -- with a double hyphen
        }
    }

    empnum decimal(6,0) {isReadOnly=yes};
    empname char(40);
end

```

- The SQL record is as follows--

```
emp Employee;
```

For further details on SQL records and implicit statements, see SQL support.

## Coding SQL statements

To prepare to code SQL statements, declare variables:

```
empnum decimal(6,0);
empname char(40);
```

**Adding a row to an SQL table:** To prepare to add a row, assign values to variables:

```
empnum = 1;
empname = "John";
```

To add the row, associate an EGL **execute** statement with an SQL INSERT statement as follows:

```

try
  execute
    #sql{
      insert into employee (empnum, empname)
      values (:empnum, :empname)
    };
onException
  myErrorHandler(8);
end

```

**Reading a set of rows from an SQL table:** To prepare to read a set of rows from an SQL table, identify a record key:

```
empnum = 1;
```

To get the data, code a series of EGL statements:

- To select a result set, run an EGL open statement--

```

open selectEmp
with #sql{
  select empnum, empname
  from employee
  where empnum >= :empnum
  for update of empname
}
into empnum, empname;

```

- To access the next row of the result set, run an EGL get next statement--

```
get next from selectEmp;
```

If you did not specify the into clause in the open statement, you need to specify the into clause in the get next statement; and if you specified the into clause in both places, the clause in the get next statement takes precedence:

```

    get next from selectEmp
    into empnum, empname;

```

The cursor is closed automatically when the last record is read from the result set.

A more complete example is as following code, which updates a set of rows:

```

VGVar.handleHardIOErrors = 1;

try
  open selectEmp
  with #sql{
    select empnum, empname
    from employee
    where empnum >= :empnum
    for update of empname
  }
  into empnum, empname;
onException
  myErrorHandler(6); // exits program
end

try
  get next from selectEmp;
onException
  if (sqlcode != 100)
    myErrorHandler(8); // exits program
  end
end

while (sqlcode != 100)
  empname = empname + " " + "III";

  try
    execute
    #sql{
      update employee
      set empname = :empname
      where current of selectEmp
    };
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next from selectEmp;
  onException
    if (sqlcode != 100)
      myErrorHandler(8); // exits program
    end
  end
end // end while; cursor is closed automatically
// when the last row in the result set is read

sysLib.commit();

```

Instead of coding the get next and while statements, you can use the `forEach` statement, which executes a block of statements for each row in a result set:

```

VGVar.handleHardIOErrors = 1;

try
  open selectEmp
  with #sql{
    select empnum, empname
    from employee
    where empnum >= :empnum

```

```

        for update of empname
    }
    into empnum, empname;
onException
    myErrorHandler(6);    // exits program
end

try
    foreach (from selectEmp)
        empname = empname + " " + "III";

        try
            execute
            #sql{
                update employee
                set empname = :empname
                where current of selectEmp
            };
        onException
            myErrorHandler(10); // exits program
        end
    end // end foreach; cursor is closed automatically
        // when the last row in the result set is read

onException
    // the exception block related to foreach is not run if the condition
    // is "sqlcode = 100", so avoid the test "if (sqlcode != 100)"
    myErrorHandler(8); // exits program
end

sysLib.commit();

```

## Using SQL records with implicit SQL statements

To begin using EGL SQL records, declare an SQL record part:

```

Record Employee type sqlRecord
{
    tableNames = ["employee"],
    keyItems = ["empnum"],
    defaultSelectCondition =
        #sqlCondition{
            aTableColumn = 4 -- start each SQL comment
                           -- with a double hyphen
        }
}

empnum decimal(6,0) {isReadOnly=yes};
empname char(40);
end

```

Declare a record that is based on the record part:

```
emp Employee;
```

**Adding a row to an SQL table:** To prepare to add a row to an SQL table, place values in the EGL record:

```
emp.empnum = 1;
emp.empname = "John";
```

Add an employee to the table by specifying the EGL add statement:

```

try
    add emp;
onException
    myErrorHandler(8);
end

```

**Reading rows from an SQL table:** To prepare to read rows from an SQL table, identify a record key:

```
emp.empnum = 1;
```

Get a single row in either of these ways:

- Specify the EGL get statement in a way that generates a series of statements (DECLARE cursor, OPEN cursor, FETCH row, and in the absence of forUpdate, CLOSE cursor):

```
try
  get emp;
onException
  myErrorHandler(8);
end
```

- Specify the EGL get statement in a way that generates a single SELECT statement:

```
try
  get emp singleRow;
onException
  myErrorHandler(8);
end
```

Process multiple rows in either of these ways:

- Use the EGL open, get next, and while statements--

```
VGVar.handleHardIOErrors = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(6);    // exits program
end

try
  get next emp;
onException
  if (emp not noRecordFound)
    myErrorHandler(8); // exit the program
  end
end

while (emp not noRecordFound)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    replace emp;
onException
  myErrorHandler(10); // exits program
end

  try
    get next emp;
onException
  if (emp not noRecordFound)
    myErrorHandler(8); // exits program
  end
end

end // end while; cursor is closed automatically
// when the last row in the result set is read

sysLib.commit();
```

- Use the EGL open and forEach statements:

```

VGVar.handleHardIOErrors = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(6);    // exits program
end

try
  forEach (from selectEmp)
    myRecord.empname = myRecord.empname + " " + "III";

    try
      replace emp;
    onException
      myErrorHandler(10);  // exits program
    end
  end // end forEach; cursor is closed automatically
      // when the last row in the result set is read

onException

  // the exception block related to forEach is not run if the condition
  // is noRecordFound, so avoid the test "if (not noRecordFound)"
  myErrorHandler(8); // exit the program
end

sysLib.commit();

```

## Using SQL records with explicit SQL statements

Before using SQL records with explicit SQL statements, you declare an SQL record part. This part is different from the previous one, in the syntax for SQL item properties and in the use of a calculated value:

```

Record Employee type sqlRecord
{
  tableNameVariables = [{"empTable"}],
                      // use of a table-name variable
                      // means that the table is specified
                      // at run time
  keyItems = ["empnum"]
}
empnum decimal(6,0) { isReadOnly = yes };
empname char(40);

// specify properties of a calculated column
aValue decimal(6,0)
{ isReadOnly = yes,
  column = "(empnum + 1) as NEWNUM" };
end

```

Declare variables:

```

emp Employee;
empTable char(40);

```

**Adding a row to an SQL table:** To prepare to add a row to an SQL table, place values in the EGL record and in a table name variable:

```

emp.empnum = 1;
emp.empname = "John";
empTable = "Employee";

```

Add an employee to the table by specifying the EGL add statement and modifying the SQL statement:



```

// a colon does not precede a table name variable
try
  add emp
  with #sql{
    insert into empTable (empnum, empname)
    values (:empnum, :empname || ' ' || 'Smith')
  }

onException
  myErrorHandler(8);
end

```

**Reading rows from an SQL table:** To prepare to read rows from an SQL table, identify a record key:

```
emp.empnum = 1;
```

Get a single row in any of these ways:

- Specify the EGL get statement in a way that generates a series of statements (DECLARE cursor, OPEN cursor, FETCH row, CLOSE cursor):

```

try
  get emp into empname // The into clause is optional. (It
                        // cannot be in the SELECT statement.)

  with #sql{
    select empname
    from empTable
    where empnum = :empnum + 1
  }

onException
  myErrorHandler(8);
end

```

- Specify the EGL get statement in a way that generates a single SELECT statement:

```

try
  get emp singleRow // The into clause is derived
                   // from the SQL record and is based
                   // on the columns in the select clause

  with #sql{
    select empname
    from empTable
    where empnum = :empnum + 1
  }

onException
  myErrorHandler(8);
end

```

Process multiple rows in either of these ways:

- Use the EGL open, get next, and while statements:

```

try

// The into clause is derived
// from the SQL record and is based
// on the columns in the select clause
open selectEmp forUpdate
  with #sql{
    select empnum, empname
    from empTable
    where empnum >= :empnum
    order by NEWNUM -- uses the calculated value
    for update of empname
  } for emp;

onException
  myErrorHandler(8); // exits the program

```

```

end

try
    get next emp;
onException
    myErrorHandler(9);    // exits the program
end

while (emp not noRecordFound)
    try
        replace emp
        with #sql{
            update :empTable
            set empname = :empname || ' ' || 'III'
        } from selectEmp;

    onException
        myErrorHandler(10); // exits the program
    end

    try
        get next emp;
    onException
        myErrorHandler(9); // exits the program
    end
end // end while

// no need to say "close emp;" because emp
// is closed automatically when the last
// record is read from the result set or
// (in case of an exception) when the program ends

sysLib.commit();

```

- Use the EGL open and forEach statements:

```

try

    // The into clause is derived
    // from the SQL record and is based
    // on the columns in the select clause
    open selectEmp forUpdate
    with #sql{
        select empnum, empname
        from empTable
        where empnum >= :empnum
        order by NEWNUM    -- uses the calculated value
        for update of empname
    } for emp;

onException
    myErrorHandler(8);    // exits the program
end

try
    forEach (from selectEmp)

        try
            replace emp
            with #sql{
                update :empTable
                set empname = :empname || ' ' || 'III'
            } from selectEmp;

        onException
            myErrorHandler(9); // exits program
        end
    end
end

```

```

end    // end forEach statement, and there is
      // no need to say "close emp;" because emp
      // is closed automatically when the last
      // record is read from the result set or
      // (in case of an exception) when the program ends

onException
  // the exception block related to forEach is not run if the condition
  // is noRecordFound, so avoid the test "if (not noRecordFound)"
  myErrorHandler(9); // exits program
end

sysLib.commit();

```

## Using EGL prepare statements

You have the option to use an SQL record part when coding the EGL prepare statement. Declare the following part:

```

Record Employee type sqlRecord
{
  tableNames = ["employee"],
  keyItems = ["empnum"],
  defaultSelectCondition =
    #sqlCondition{
      aTableColumn = 4 -- start each SQL comment
                      -- with a double hyphen
    }
}

empnum decimal(6,0) {isReadOnly=yes};
empname char(40);
end

```

Declare variables:

```

emp Employee;
empnum02 decimal(6,0);
empname02 char(40);
myString char(120);

```

**Adding a row to an SQL table:** Before adding a row, assign values to variables:

```

emp.empnum = 1;
emp.empname = "John";
empnum02 = 2;
empname02 = "Jane";

```

Develop the SQL statement:

- Code the EGL prepare statement and reference an SQL record, which provides an SQL statement that you can customize:

```

prepare myPrep
  from "insert into employee (empnum, empname) " +
      "values (?, ?)" for emp;

// you can use the SQL record
// to test the result of the operation
if (emp is error)
  myErrorHandler(8);
end

```

- Alternatively, code the EGL prepare statement without reference to an SQL record:

```

myString = "insert into employee (empnum, empname) " +
  "values (?, ?)";

try

```

```

        prepare addEmployee from myString;
onException
    myErrorHandler(8);
end

```

In each of the previous cases, the EGL prepare statement includes placeholders for data that will be provided by an EGL execute statement. Two examples of the execute statement are as follows:

- You can provide values from a record (SQL or otherwise):  

```
execute addEmployee using emp.empnum, emp.empname;
```
- You can provide values from individual items:  

```
execute addEmployee using empnum02, empname02;
```

**Reading rows from an SQL table:** To prepare to read rows from an SQL table, identify a record key:

```
empnum02 = 2;
```

You can replace multiple rows in either of these ways:

- Use the EGL open, while, and get next statements--  

```

myString = "select empnum, empname from employee " +
           "where empnum >= ? for update of empname";

try
    prepare selectEmployee from myString for emp;
onException
    myErrorHandler(8);    // exits the program
end

try
    open selectEmp with selectEmployee
        using empnum02
        into emp.empnum, emp.empname;
onException
    myErrorHandler(9);    // exits the program
end

try
    get next from selectEmp;
onException
    myErrorHandler(10);   // exits the program
end

while (emp not noRecordFound)

    emp.empname = emp.empname + " " + "III";

    try
        replace emp
        with #sql{
            update employee
            set empname = :empname
        }
        from selectEmp;
onException
    myErrorHandler(11); // exits the program
end

try
    get next from selectEmp;
onException
    myErrorHandler(12); // exits the program

```

```

        end
    end // end while; close is automatic when last row is read

    sysLib.commit();

```

- Use the EGL open and forEach statements--
 

```

myString = "select empnum, empname from employee " +
           "where empnum >= ? for update of empname";

try
    prepare selectEmployee from myString for emp;
onException
    myErrorHandler(8);    // exits the program
end

try
    open selectEmp with selectEmployee
        using empnum02
        into emp.empnum, emp.empname;
onException
    myErrorHandler(9);    // exits the program
end

try
    forEach (from selectEmp)
        emp.empname = emp.empname + " " + "III";

        try
            replace emp
            with #sql{
                update employee
                set empname = :empname
            }
            from selectEmp;
onException
    myErrorHandler(11); // exits the program
end
end // end forEach; close is automatic when last row is read
onException

// the exception block related to forEach is not run if the condition
// is noRecordFound, so avoid the test "if (not noRecordFound)"
myErrorHandler(12); // exits the program
end

sysLib.commit();

```

## Default database

The default database is a relational database that is accessed when an SQL-related I/O statement runs in EGL-generated code and when no other database connection is current. The default database is available from the beginning of a run unit; however, you can dynamically connect to a different database for subsequent access in the same run unit, as described in *VGLib.connectionService*.

The default database is specified in the optional runtime property **vgj.jdbc.default.database**, which receives a generated value from build descriptor option **sqlDB** if option **genProperties** is set to GLOBAL or PROGRAM at generation time.

### Related concepts

“Java runtime properties” on page 431  
 “Run unit” on page 866  
 “SQL support” on page 277

#### Related tasks

"Setting up a J2EE JDBC connection" on page 445

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

"Understanding how a standard JDBC connection is made" on page 309

#### Related reference

"Java runtime properties (details)" on page 642

"sqlDB" on page 490

"connectionService()" on page 1047

## Informix and EGL

The following rules are specific to Informix databases and EGL:

- An Informix database that is accessed by EGL or by an EGL-generated program must have transactions enabled.
- If you are coding an SQL statement and use a colon (:) when identifying an Informix table, use quote marks to separate the Informix identifier from the rest of the statement, as in these examples:

```
INSERT INTO "myDB:myTable"  
      (myColumn) values (:myField)
```

```
INSERT INTO "myDB@myServer:myTable"  
      (myColumn) values (:myField)
```

- If you are using the SQL retrieve feature of EGL to access data from a non-ANSI Informix database, make sure that any database column of type DECIMAL includes a scale value. Instead of defining a column as DECIMAL (4), for example, define the column as DECIMAL (4,0).
- If you intend to use the SQL retrieve feature to retrieve data from a table that is part of an Informix system schema, you must set a special preference, as described in *Setting preferences for SQL retrieve*.

#### Related concepts

"SQL support" on page 277

#### Related tasks

"Retrieving SQL table data"

"Setting preferences for SQL retrieve" on page 123

---

## SQL-specific tasks

### Retrieving SQL table data

EGL provides a way to create SQL record items from the definition of an SQL table, view, or join; for an overview, see *SQL support*.

Do as follows:

1. Ensure that you have set SQL preferences as appropriate. For details, see *Setting preferences for SQL retrieve*.
2. Decide where to do the task--
  - In an EGL source file, as you develop each SQL record; or
  - In the Outline view, as may be easier when you already have SQL records.
3. If you are working in the EGL source file, proceed in this way--
  - a. If you do not have the SQL record, create it:

- 1) Type **R**, press Ctrl-Space, and in the content-assist list, select one of the SQL table entries (usually **SQL record with table names**).
- 2) Type the name of the SQL record; press Tab; and type a table name, or a comma-delimited list of tables, or the alias of a view.

You also can create an SQL record by typing the minimal content, as appropriate if the name of the record is the same as the name of the table, as in this example:

```
Record myTable type sqlRecord
end
```

- b. Right-click anywhere in the record.
  - c. In the context menu, click **SQL record > Retrieve SQL**.
4. If you are working in the Outline view, right click on the entry for the SQL record and, in the context menu, click **Retrieve SQL**.

**Note:** You cannot retrieve an SQL view that is defined with the DB2 condition **WITH CHECK OPTIONS**.

After you create record items, you may want to gain a productivity benefit by creating the equivalent dataItem parts; see *Overview on creating dataItem parts from an SQL record part*.

#### Related concepts

“Creating dataItem parts from an SQL record part (overview)”  
 “SQL support” on page 277

#### Related tasks

“Creating dataItem parts from an SQL record part” on page 301  
 “Setting preferences for SQL database connections” on page 121  
 “Setting preferences for SQL retrieve” on page 123

#### Related reference

“SQL item properties” on page 68

## Creating dataItem parts from an SQL record part (overview)

After you declare structure items in an SQL record part, you can use a special mechanism in the EGL editor to create DataItem parts that are equivalent to the structure items. The benefit is that you can more easily create a non-SQL record (usually a basic record) for transferring data to and from the related SQL record at run time.

Consider the following structure items:

```
10 myHostVar01 CHAR(3);
10 myHostVar02 BIN(9,2);
```

You can request that dataItem parts be created:

```
DataItem myHostVar01 CHAR(3) end

DataItem myHostVar02 BIN(9,2) end
```

Another effect is that the structure item declarations are rewritten:

```
10 myHostVar01 myHostVar01;
10 myHostVar02 myHostVar02;
```

As shown in this example, each `dataItem` part is given the same name as the related structure item and acts as a typedef for the structure item. Each `DataItem` part is also available as a typedef for other structure items.

Before you can use a structure item as the basis of a `dataItem` part, the structure item must have a name, must have valid primitive characteristics, and must not point to a typedef.

#### **Related concepts**

"SQL support" on page 277

#### **Related tasks**

"Creating `dataItem` parts from an SQL record part"

#### **Related reference**

"`DataItem` part in EGL source format" on page 566

"SQL record part in EGL source format" on page 877

### **Creating `dataItem` parts from an SQL record part**

After you declare structure items in an SQL record part, you can use a special mechanism in the EGL editor to create `dataItem` parts that are equivalent to the structure items. For general information, see *Overview on creating `dataItem` parts from an SQL record part*.

If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.

Do as follows in the Outline view:

1. For a given SQL record part, hold down **Ctrl** while clicking on each of the structure items of interest. To select all the structure items in a given record, click the topmost structure item, then hold down **Shift** while clicking on the bottommost structure item.
2. Right-click on the selected structure items.
3. In the context menu, click **Create DataItem part**.

The `dataItem` parts are written at the bottom of the EGL source file, and each structure item is changed to refer to the equivalent part.

#### **Related concepts**

"Creating `dataItem` parts from an SQL record part (overview)" on page 300

"SQL support" on page 277

#### **Related tasks**

"Retrieving SQL table data" on page 299

#### **Related reference**

"SQL record part in EGL source format" on page 877

## **Creating EGL data parts from relational database tables**

### **EGL Data Parts wizard**

The EGL Data Parts wizard lets you create SQL record parts, as well as related `dataItem` parts and library-based function parts, from one or more relational database tables or pre-existing views.



After connecting to the database, you can do as follows:

- Specify the SQL-record key fields that are used to create, read, update, or delete a row from a given database table or view.
- Customize explicit SQL statements for creating, reading, or updating a row. (The SQL statement for deleting a row cannot be customized.)
- Specify the SQL-record key fields that are used to select a set of rows from a given database or view.
- Customize an explicit SQL statement for selecting a set of rows.
- Validate and run each SQL statement

The output includes these files:

- An EGL source file that defines each record part
- An EGL library for each record part
- An EGL source file that contains all the dataItem parts referenced by the structure items in the SQL record parts

You can reduce the number of files if you select the **Record and library in the same file** check box.

#### Related concepts

“SQL support” on page 277

#### Related tasks

“Creating, editing, or deleting a database connection for the EGL wizards” on page 303

“Creating EGL data parts from relational database tables”

“Customizing SQL statements in the EGL wizards” on page 304

### Creating EGL data parts from relational database tables

To create EGL data parts from relational database tables without creating a separate Web application, do as follows:

1. Select **File > New > Other...** A dialog is displayed for selecting a wizard.
2. Expand **EGL** and double-click **EGL Data Parts**. The EGL Data Parts dialog is displayed.
3. Enter an EGL or EGL Web project name, or select an existing project from the drop-down list. The parts will be generated into this project.
4. Select an existing database connection from the drop-down list or establish a new database connection--
  - To establish a new database connection, click **Add** and interact with the *New Database Connection Wizard*. For details on the kind of input required in a particular field, right click into the field and press F1.
  - For details on editing or deleting a database connection, see *Creating, editing, or deleting a database connection for the EGL wizards*.

When a connection is made to the database, a list of database tables is displayed.

5. If you do not want to accept the default EGL file name for data items, type a new file name.
6. In the **Select your data** field, click on the name of the table whose columns will help you to declare data parts. To select multiple tables, hold down the **Ctrl** key while clicking on different table names. To transfer the highlighted name or names to the list of selected tables, click the right arrow.

7. For each of the selected tables (on the right), either specify the name of the EGL record to be created or accept the default name. To remove one or more tables from that list, highlight the entries of interest and click the left arrow.
8. If you want to include the library part and SQL record parts in the same file, select the check box.
9. Click **Next**.
10. A tab is available for each table. In each tab, select the key field to use when reading, updating, and deleting individual rows, then click the right arrow. To select multiple key fields, hold down the **Ctrl** key while clicking on different field names. To remove a key field from the list on the right, highlight the field name and click the left arrow.
11. Choose the selection condition field to use when selecting a set of rows, then click the right arrow. To select multiple fields, hold down the **Ctrl** key while clicking on different field names. To remove a field from the list on the right, highlight the field name and click the left arrow.
12. To customize an implicit SQL statement, see *Customizing SQL statements in the EGL wizards*. This option is not available for the EGL delete statement.
13. Click **Next**.
14. The Generate EGL Data Parts screen is displayed, including (at the bottom) a list of the files that will be produced:
  - a. To change the name of the EGL project that will receive the EGL parts, type a project name in the Destination project field or select a project from the related drop-down list.
  - b. To specify the EGL packages for a specific type of part (data or library), type a package name in the related field or select a name from the related drop-down list.
15. Click **Finish**.

#### Related concepts

"EGL Data Parts wizard" on page 301

"EGL Data Parts and Pages wizard" on page 217

"SQL support" on page 277

#### Related tasks

"Creating a single-table EGL Web application" on page 218

"Creating, editing, or deleting a database connection for the EGL wizards"

"Customizing SQL statements in the EGL wizards" on page 304

**Creating, editing, or deleting a database connection for the EGL wizards:** When you are at the first screen in an EGL wizard for creating data parts from a relational database table or for creating a Web application from a relational database table, you specify a database connection in either of two ways:

- Select an existing connection from a drop-down list; or
- Interact with the *New Database Connection Wizard*.

To use that wizard to create a connection, click **Add** and add information as required. For details on the kind of input required in a particular field, left click into the field and press F1.

To edit an existing database connection, do as follows:

1. Select **Window > Open Perspective > Other**. At the Select Perspective dialog, select the **Show all** check box and double-click **Data**.

2. In the Database Explorer view, right-click on the database connection, then select **Edit Connection**. Step through the pages of the database connection wizard and change information as appropriate. For help, press F1.
3. To complete the edit, click **Finish**.

To delete an existing database connection, do as follows:

1. Select **Window > Open Perspective > Other**. At the Select Perspective dialog, select the **Show all** check box and double-click **Data**.
2. In the Database Explorer view, right-click on the database connection, then select **Delete**.

#### Related concepts

"EGL Data Parts wizard" on page 301

"EGL Data Parts and Pages wizard" on page 217

"SQL support" on page 277

#### Related tasks

"Creating a single-table EGL Web application" on page 218

"Creating EGL data parts from relational database tables" on page 302

"Setting EGL preferences" on page 115

**Customizing SQL statements in the EGL wizards:** When you are using an EGL wizard to create data parts from a relational table or create a Web application from a relational table, you can change the SQL statement that is associated with an action like read or update:

1. Select an action from the Edit actions list, then click **Edit SQL**.
2. Edit the SQL statement (as is possible for all actions except delete), then click **Validate**. Validation ensures that the statement has the correct syntax and adheres to the rules for host variable names. If the statement contains errors, a message is displayed. Correct the errors and validate again.  
**Revert to Last** changes the statement into its last valid modified version. Previous versions become unavailable after you close the dialog.
3. Click **Execute**, then click **Execute** again.
4. If the SQL statement requires values for host variables, the Specify Variable Values dialog is displayed. Double-click the **Value** field to enter the value of a host variable, then press the **Enter** key. When you have entered values for all host variables, click **Finish**.

**Note:** For host variables defined as type *character*, you must enclose the value in single quotes.

5. When you are finished executing the SQL statement, click **Close**.
6. When you are finished editing the SQL statements, click **OK**.

#### Related concepts

"EGL Data Parts wizard" on page 301

"EGL Data Parts and Pages wizard" on page 217

"SQL support" on page 277

#### Related tasks

"Creating a single-table EGL Web application" on page 218

"Creating EGL data parts from relational database tables" on page 302

"Creating, editing, or deleting a database connection for the EGL wizards" on page 303

"Setting EGL preferences" on page 115

## Viewing the SQL SELECT statement for an SQL record

EGL provides an implicit SQL SELECT statement for a given SQL record part. To view the implicit SQL SELECT statement, do as follows:

1. Open the EGL file that contains the SQL record part. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With > EGL Editor**.
2. Click inside the SQL record part, then right-click. A context menu displays.
3. Select **SQL Record > View Default Select**.
4. To validate the SQL SELECT statement against a database, click **Validate**.

**Note:** Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

### Related concepts

"SQL support" on page 277

### Related tasks

"Validating the SQL SELECT statement for an SQL record"

### Related reference

"SQL record part in EGL source format" on page 877

## Validating the SQL SELECT statement for an SQL record

EGL provides an implicit SQL SELECT statement for a given SQL record part. To validate the implicit SQL SELECT statement against a database, do as follows:

1. Open the EGL file that contains the SQL record part. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With > EGL Editor**.
2. Click inside the SQL record part, then right-click. A context menu displays.
3. Select **SQL Record > Validate Default Select**.

**Note:** Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

### Related concepts

"SQL support" on page 277

### Related tasks

"Viewing the SQL SELECT statement for an SQL record"

### Related reference

"SQL record part in EGL source format" on page 877

## Constructing an EGL prepare statement

Within a function, you can construct the following kinds of EGL statements that are based on an SQL record part:

- An EGL **prepare** statement; and
- The related EGL **execute**, **open**, or **get** statement.

Do as follows:

1. Open an EGL file with the EGL editor. The file must contain a function and a coded SQL statement. If you do not have a file open, right-click on the EGL file in the workbench Project Explorer, then select **Open With > EGL Editor**.
2. Click inside the function at the location where the EGL **prepare** statement will reside, then right-click. A context menu displays.
3. Select **Add SQL Prepare Statement**.
4. Type a name to identify the EGL **prepare** statement. For rules, see *Naming conventions*.
5. If you have an SQL record variable defined, select it from the drop-down list. The corresponding SQL record part name displays. If you do not have an SQL record variable defined, you can type a name in the SQL record variable name field, then select an SQL record part name using the **Browse** button. You must eventually define an SQL record variable with that name in the EGL source code.
6. Select an execution statement type from the drop-down list.
7. If the execution statement is of type open, enter a result-set identifier.
8. Click **OK**. EGL statements are constructed inside the function.

### Related concepts

"SQL support" on page 277

### Related tasks

"Validating the SQL SELECT statement for an SQL record" on page 305

"Viewing the SQL SELECT statement for an SQL record" on page 305

### Related reference

"Naming conventions" on page 778

"SQL record part in EGL source format" on page 877

## Constructing an explicit SQL statement from an implicit one

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. To construct an explicit SQL statement from an implicit one, do as follows:

1. Open the EGL file that contains the EGL I/O statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With > EGL Editor**.
2. Click on the EGL I/O statement, then right-click. A context menu displays.
3. To construct an explicit SQL statement without an INTO clause, select **SQL Statement > Add**. To construct an explicit SQL statement with an INTO clause, select **SQL Statement > Add with Into**. The implicit SQL statement is appended to the EGL I/O statement making it an explicit SQL statement.

**Note:** The INTO clause is only valid with **open**, **get**, and **get next** statements.

### Related concepts

"SQL support" on page 277

### Related tasks

"Removing an SQL statement from an SQL-related EGL statement" on page 308

"Resetting an explicit SQL statement" on page 308

"Validating an implicit or explicit SQL statement"

"Viewing the implicit SQL for an SQL-related EGL statement"

## Viewing the implicit SQL for an SQL-related EGL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. To view the implicit SQL for an EGL I/O statement, do as follows:

1. Open the EGL file that contains the EGL I/O statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With > EGL Editor**.
2. Click on the EGL I/O statement, then right-click. A context menu displays.
3. Select **SQL Statement > View**.

### Related concepts

"SQL support" on page 277

### Related tasks

"Constructing an explicit SQL statement from an implicit one" on page 306

"Removing an SQL statement from an SQL-related EGL statement" on page 308

"Resetting an explicit SQL statement" on page 308

"Validating an implicit or explicit SQL statement"

## Validating an implicit or explicit SQL statement

To validate an implicit or explicit SQL statement against a database, do as follows:

1. Open the EGL file that contains the SQL-related EGL statement or explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With > EGL Editor**.
2. Click the EGL statement or SQL statement, then right-click. A context menu displays.
3. Select **SQL Statement > Validate**.

**Note:** Before using the validate function, DB2 UDB users must set the DEFERREDPREPARE option. You can set this option interactively in the CLP (DB2 command line processor) using the **db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** command. This command will put the keyword under the COMMON section. Execute the command **db2 get cli cfg for section common** to verify that the keyword is being picked up.

### Related concepts

"SQL support" on page 277

### Related tasks

"Constructing an explicit SQL statement from an implicit one" on page 306

"Removing an SQL statement from an SQL-related EGL statement" on page 308

"Resetting an explicit SQL statement" on page 308

"Viewing the implicit SQL for an SQL-related EGL statement"

## Resetting an explicit SQL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. An implicit SQL statement can be appended to an EGL I/O statement making it an explicit SQL statement. If you change the explicit SQL statement, do as follows to return to an SQL statement based on the implicit SQL:

1. Open the EGL file that contains the explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With > EGL Editor**.
2. Click on the explicit SQL statement, then right-click. A context menu displays.
3. Select **SQL Statement > Reset**.

### Related concepts

"SQL support" on page 277

### Related tasks

"Constructing an explicit SQL statement from an implicit one" on page 306

"Removing an SQL statement from an SQL-related EGL statement"

"Validating an implicit or explicit SQL statement" on page 307

"Viewing the implicit SQL for an SQL-related EGL statement" on page 307

## Removing an SQL statement from an SQL-related EGL statement

EGL provides an implicit SQL statement for each SQL-related EGL input/output (I/O) statement. An implicit SQL statement can be appended to an EGL I/O statement making it an explicit SQL statement (see *Constructing an explicit SQL statement from an implicit one*). To remove the appended SQL statement, do as follows:

1. Open the EGL file that contains the explicit SQL statement. If you do not have the file open, right-click on the EGL file in the Project Explorer, then select **Open With > EGL Editor**.
2. Click on the explicit SQL statement, then right-click. A context menu displays.
3. Select **SQL Statement > Remove**. The EGL I/O statement remains.

### Related concepts

"SQL support" on page 277

### Related tasks

"Constructing an explicit SQL statement from an implicit one" on page 306

"Resetting an explicit SQL statement"

"Validating an implicit or explicit SQL statement" on page 307

"Viewing the implicit SQL for an SQL-related EGL statement" on page 307

## Resolving a reference to display an implicit SQL statement

Consider what happens when you specify the following EGL statement:

```
open myRecord;
```

When the EGL editor tries to create a default SQL statement, the editor attempts to find a variable named myRecord and to identify the SQL record part on which that variable is based. If the variable is unavailable at development time or if the variable is undeclared, the editor attempts to use an SQL record part named myRecord as the basis for the default SQL statement. The editor assumes that you intend to create a variable whose name is the name of the SQL record part.



If you wish to store an SQL-related function in a file that does not include the variable `myRecord`, you can do as follows:

1. In the program part, declare the global variable
2. Create the function as a nested function in the program part
3. Create the default SQL statement and modify it as appropriate; then, save the file
4. Move the function to the other file

After the function is moved from the program part, the record name cannot be resolved at development time, and the editor cannot display any default SQL statements that are based on that record.

#### Related concepts

“SQL support” on page 277

## Understanding how a standard JDBC connection is made

A standard JDBC connection is created for you at run time if you are debugging a generated Java program and if the program properties file includes the necessary values. For details on the meaning of the program properties, including details on how the values are derived, see *Java runtime properties (details)*.

The JDBC connection is based on the following kinds of information:

#### Connection URL

If your code tries to access a database before invoking the system function `sysLib.connect` or `VGLib.connectionService`, the connection URL is the value of property `vgj.jdbc.default.database`.

If your code tries to access a database in response to an invocation of the system function `sysLib.connect` or `VGLib.connectionService`, the connection URL is the value of property `vgj.jdbc.databaseSN`.

For details on the format of a connection URL, see *sqlValidationConnectionURL*.

#### User ID

If your code tries to access a database before invoking the system function `sysLib.connect` or `VGLib.connectionService`, the user ID is the value of property `vgj.jdbc.default.userid`.

If your code tries to access a database in response to an invocation of one of those system functions, the user ID is a value specified in the invocation.

#### Password

If your code tries to access a database before invoking the system function `sysLib.connect` or `VGLib.connectionService`, the password is the value of property `vgj.jdbc.default.password`.

If your code tries to access a database in response to an invocation of one of those system functions, the password is a value specified in the invocation. You can use a system function to avoid exposing the password in the program properties file.

#### JDBC driver class

The JDBC driver class is the value of property `vgj.jdbc.drivers`.

#### Related concepts

“Program properties file” on page 433



### Related tasks

"Setting up a J2EE JDBC connection" on page 445

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

### Related reference

"connect()" on page 1025

"connectionService()" on page 1047

"genProperties" on page 480

"Java runtime properties (details)" on page 642

"JDBC driver requirements in EGL" on page 660

"sqlDB" on page 490

"sqlID" on page 491

"sqlPassword" on page 492

"sqlValidationConnectionURL" on page 493

"sqlJDBCClass" on page 491

---

## DL/I database support

EGL-generated code can access a Data Language/I (DL/I) database on any of following target systems:

- CICS for z/OS
- IMS BMP
- IMS/VS
- z/OS batch

EGL allows this access in two ways:

- Through EGL I/O keywords such as **add**, **get**, and **replace**, which generate DL/I statements in the output COBOL code. The default DL/I statements that EGL generates from EGL keywords are called *implicit* DL/I statements. (You have the option to customize these statements.)
- By giving you the ability to code *explicit* DL/I statements yourself. You may code explicit statements in two ways:
  - By customizing an EGL I/O keyword statement through the **with #dli{ statement }** syntax. For more information, see *#dli directive*.
  - By using the library functions `DLILib.AIBTDLI()`, `DLILib.EGLTDLI()`, or `VGLib.VGTDLI()` to code DL/I calls. See *DLILib* and *VGLib*.

Note that the implicit and explicit DL/I code uses a powerful pseudo-DL/I syntax with some differences from actual DL/I code. For more information, see *#dli directive*.

If you are not familiar with DL/I, see *Basic DL/I concepts*.

## EGL keywords and DL/I

The next table lists the EGL keywords that you can use to access a DL/I database. Included in this table are sample DL/I calls that each keyword might generate. When you code an EGL **add** statement, for example, you generate a DL/I ISRT call. You can preview an unformatted version of the DL/I calls that EGL will generate by placing your cursor in a line of code that contains a keyword and right-clicking the line. From the popup menu, select **DLI Statement > View**.

You can also code DL/I calls directly, using the **with #dli{ statement }** syntax. This syntax allows you to further specify how your program should access the database, as is necessary if the default DL/I calls that EGL generates from I/O keywords are not sufficient.

The following table shows the EGL I/O statements that support DL/I databases, and some examples of COBOL code those EGL statements might generate. The examples assume you have used the **@DLI** complex property to set your call interface to CBLTDLI. You can just as easily use the default AIBTDLI interface, in which case the COBOL will contain an AIB mask instead of a PCB mask. The AIB mask allows the program to specify a PCB by name rather than address. For details on the **@DLI** property, see **@DLI**.

I/O keyword/purpose	Example DL/I call
<p>“add” on page 661</p> <p>Places a record of type DLIsegment into a DL/I database; or (if you use a dynamic array of DLIsegment records), places a set of records under a single parent segment, based on the content of each element of the array.</p>	<p>CALL 'CBLTDLI' USING EZEDLI-ISRT EZPCB-MASK01 EZEORSEG-IO-AREA EZECUST-QUALIFIED-SSA EZELOC-QUALIFIED-SSA EZEORSSA-EZEUNQ-UNQUALIFIED-SSA</p>
<p>“delete” on page 673</p> <p>Deletes a segment from a database. You must first use a <b>get...forUpdate</b>, <b>get next...forUpdate</b>, or <b>get next inParent...forUpdate</b> statement (generating a DL/I GHU, GHN, or GHNP statement) to locate and hold the record.</p>	<p>CALL 'CBLTDLI' USING EZEDLI-DLET EZPCB-MASK01 EZEORSEG-IO-AREA</p>
<p>“get” on page 687</p> <p>Reads a unique segment from a database; or (if you use a dynamic array of DLIsegment records), reads successive segments into successive elements in the array. If you use <b>get</b> to read successive records into an array, the COBOL program will loop through a series of GN calls following the initial GU.</p>	<p>CALL 'CBLTDLI' USING EZEDLI-GU EZPCB-MASK01 EZEORSEG-IO-AREA EZECUST-QUALIFIED-SSA EZELOC-QUALIFIED-SSA</p>
<p>get...forUpdate</p> <p>Reads and holds a unique segment from a database; or (if you use a dynamic array of DLIsegment records), reads successive segments into successive elements in the array and holds them (by issuing a series of GHN calls after the initial GHU).</p>	<p>CALL 'CBLTDLI' USING EZEDLI-GHU EZPCB-MASK01 EZEORSEG-IO-AREA EZECUST-QUALIFIED-SSA EZELOC-QUALIFIED-SSA</p>
<p>“get next” on page 701</p> <p>Reads the next segment based on the current position in the database.</p>	<p>CALL 'CBLTDLI' USING EZEDLI-GN EZPCB-MASK01 EZEORSEG-IO-AREA EZEUNQ-UNQUALIFIED-SSA</p>

I/O keyword/purpose	Example DL/I call
get next...forUpdate  Reads and holds the next segment based on the current position in the database.	CALL 'CBLTDLI' USING EZEDLI-GHN EZPCB-MASK01 EZEORSEG-IO-AREA EZEUNQ-UNQUALIFIED-SSA
“get” on page 687  Reads the next segment that has the same parent segment, based on the current position in the database.	CALL 'CBLTDLI' USING EZEDLI-GNP EZPCB-MASK01 EZEORSEG-IO-AREA EZEUNQ-UNQUALIFIED-SSA
get next inParent...forUpdate  Reads and holds the next segment that has the same parent segment, based on the current position in the database.	CALL 'CBLTDLI' USING EZEDLI-GHNP EZPCB-MASK01 EZEORSEG-IO-AREA EZEUNQ-UNQUALIFIED-SSA
“replace” on page 738  Puts a changed segment back into a database. You must first use a <b>get...forUpdate</b> , <b>get next...forUpdate</b> , or <b>get next inParent...forUpdate</b> statement (generating a DL/I GHU, GHN, or GHNP statement) to locate and hold the record.  Under no circumstances can you update multiple database segments with a single default EGL <b>replace</b> statement.	CALL 'CBLTDLI' USING EZEDLI-REPL EZPCB-MASK01 EZEORSEG-IO-AREA

The EGL **set** keyword (specifically the *set record* position statement) also supports DL/I. This keyword performs no I/O, but effectively locates a specified segment within a database. For details on how the keyword operates, see *set*.

## Record parts for DL/I support

You create the following kinds of record parts:

### DLISegment

The DLISegment record part corresponds to a segment in a DL/I database. You create a record that is based on that part, then use that record to access the segment.

### PSBRecord

The PSBRecord part contains information on a runtime program specification block (PSB) and includes a set of records that each include details on a runtime program control block (PCB).

You create a record that is based on the PSBRecord part, then assign the record name to the program **psb** property. EGL uses the record to generate the code that creates and validates DL/I calls.

When working with DL/I, you also create records that are based on the following predefined record parts, each of which matches the layout of a runtime PCB:

### IO\_PCBRecord

A record of this type lets a program communicate with a user through a terminal

### **ALT\_PCBRecord**

A record of this type lets a program change the destination for an outgoing message

### **DB\_PCBRecord**

A record of this type allows access to a DL/I database

### **GSAM\_PCBRecord**

A record of this type is used for accessing a file by way of the Generalized Sequential Access Method (GSAM), in a program generated for z/OS batch or BMP. The file acts as a root-only DL/I database, but input and output is by way of a serial record. Commits and rollbacks affect the file.

Within each PCB record, you use the complex property **@PCB** to define the type and name of the PCB, and in some cases the hierarchy of the segments visible to that PCB. For details on the record types, see *PCB record part properties and Record types and properties*.

Finally, you may create a fixed record that is based on the predefined record part **PSBDataRecord**. You can use the record to interact with the system variable **DLILib.psbData**, which contains both the name of the runtime PSB and an address with which that PSB is accessed. The record is useful if you need to pass the PSB (really, a name and address) to another program or to receive the PSB from another program.

## **DLISegment part**

Each segment type that you want to access in a DL/I database must have an equivalent record of type **DLISegment** in your program.

Consider the sample customer database and related code in *Example DL/I database*. For each customer there are segments for credit status, history, and individual locations. Each location has order segments, and each order has line item segments. In this case you will create **DLISegment** type records for Customer, Credit, History, Location, Order, and Line Item.

For reasons of clarity, because of migration issues, or because of conflicts with EGL naming conventions, you may want to use different names in EGL than in DL/I for the segment name or for field names. You can define one or more of the following properties in a set-value block:

- **segmentName char(8)** If you did not use the DL/I segment name for the name of your **DLISegment** record, provide the segment name from the database here. For example, if you had a DL/I database that kept track of students and one of the segment types was named **TRANSFER**, you would not be able to use that name for a **DLISegment** record in EGL, as EGL uses "transfer" as a reserved keyword. Instead you could name the **DLISegment** record "xferStudents" and set **segmentName="TRANSFER"**. EGL converts all names to upper case when generating the COBOL source code, so **custName** in your EGL source is equivalent to **CUSTNAME** in COBOL.
- **hostVarQualifier String** EGL allows you to use host variables (variables defined in your EGL host program rather than in DL/I) when you construct **#dli** directives. These directives override the default DL/I calls that EGL generates from I/O keywords like **add** or **get**.

In the absence of a **#dli** directive, EGL builds default segment search arguments (SSAs) to locate specific DL/I segments. As shown in the next example, the **hostVarQualifier** property of a **DLISegment** record part identifies a record that will contain a key value used in a default SSA .

Consider the case in which the customer segment STSCCST is the parent of the location segment STSCLOC, and you want to retrieve only the data in the location segment. When you define the three record parts to set up this situation, you omit (for now) the `hostVarQualifier` property:

```
Record myCustomerRecordPart type DLISegment
{ segmentName="STSCCST", keyItem="customerNo" }
  10 customerNo char(6)      { dliFieldName = "STQCCNO" }; //key field
  ...
end

Record myLocationRecordPart type DLISegment
{ segmentName="STSCLOC", keyItem="locationNo" }
  10 locationNo char(6)      { dliFieldName = "STQCLNO" }; //key field
  ...
end

Record myCustomerPSBRecordPart type PSBRecord { defaultPSBName="STBICLG" }
// database PCB
customerPCB DB_PCBRecord { @PCB {
  pcbType = DB,
  pcbName = "STDCDBL",
  hierarchy = [
    @Relationship { segmentRecord = "myCustomerRecordPart" },
    @Relationship {
      segmentRecord = "myLocationRecordPart", parentRecord="myCustomerRecordPart" },
    ...]};
end
```

Next, you associate the program with a record that is based on the PSBRecord part and define variables based on the record parts:

```
Program myProgram {
  @DLI{ psb = "myCustomerPSB" }}

//define variables
myCustomerPSB myCustomerPSBRecordPart;
myCustomer myCustomerRecordPart;
myLocation myLocationRecordPart;
```

Finally, in a function, you attempt to retrieve the first location segment in the database without first retrieving any customer data:

```
get myLocation;
```

From this **get** statement, EGL will generate the following pseudo-DL/I code:

```
GU STSCCST (STQCCNO = :myCustomerRecordPart.customerNo)
  STSCLOC (STQCLNO = :myLocation.locationNo)
```

Notice that while EGL correctly associated the variable `myLocation` with the segment STSCLOC, EGL was unable to find the `myCustomer` variable. EGL knew only that `myLocation` is of type `myLocationRecordPart`, whose parent segment is `myCustomerRecordPart`, and that the `keyItem` for `myCustomerRecordPart` is `customerNo`.

You could solve this problem by giving your variables the same name as the record parts on which they are based. Doing so could easily create confusion, however, and preferred practice is to use different names for parts and variables. You could also code a `#dli` directive in which you copied the above pseudo-DL/I code and replaced `:myCustomerRecordPart` with `:myCustomer`. In a third, simpler approach, you can add a `hostVarQualifier` to the declaration for `myCustomerRecordPart`, as follows:

```
Record myCustomerRecordPart type DLISegment
{ segmentName="STSCCST", hostVarQualifier="myCustomer", keyItem="customerNo" }
```

Now the same **get** statement will produce the correct pseudo-DL/I code:

```
GU STSCCST (STQCCNO = :myCustomer.customerNo)
  STSCLOC (STQCLNO = :myLocation.locationNo)
```

You also could use the property `hostVarQualifier` to refer to a field in a record that is not based on a segment in the database. For example, if you want EGL to look for the key value of the customer segment in a basic record (such as a transaction record), you must assign the name of that record to the **hostVarQualifier** property.

- **lengthItem FieldReference** If the `DLISegment` record part defines a variable-length record, name the field within the record that specifies the record length.
- **keyItem FieldReference** If the database specifies a key for this segment, identify the field that contains the key value. As noted earlier, the field can be in any record you choose, but is usually in a record that is based on the `DLISegment` record part being defined. EGL uses the **keyItem** property for creating default SSAs.

If you are working with a `DLISegment` record field whose name does not match the name of the equivalent database-segment field, set the record-field property **dliFieldName** to the name of the segment field.

### PSBRecord part

The `PSBRecord` part contains information about your program's access to one or more databases and is used to provide access to the runtime PSB. Each record of that part type includes the following field:

- **defaultPSBName char(8)**. Identifies the runtime PSB by name. EGL assigns the value of **defaultPSBName** to the field **psbName** is the system variable **DLILib.psbData**. In CICS, that assignment causes the program to use the specified runtime PSB. The default value of **defaultPSBName** is the name of the `PSBRecord` part.

### PSBDataRecord part

This predefined record part is the type of data that is found in the system variable **DLILib.psbData**. A record that is based on the `PSBDataRecord` part contains the following fields:

#### **psbName char(8)**

CICS uses this name to determine the next PSB to schedule. IMS does not use this name.

#### **psbRef int**

The field **psbRef** provides access to the list of runtime PCBs. Do not modify this field directly.

In CICS, you can schedule the PSB identified in **psbName**; and the scheduling occurs after **psbRef** is cleared by a commit. To use this capability, set **psbRef** to a 0.

### Declaring record parts

The following example shows one way to declare a `DLISegment` record that reflects a segment in a DL/I database:

```
Record myCustomerRecordPart type DLISegment
{ segmentName="STSCCST", keyItem="customerNo" }
  10 customerNo char(6)      { dliFieldName = "STQCCNO" }; //key field
  10 customerName char(25)   { dliFieldName = "STUCCNM" };
  10 customerAddr1 char(25)  { dliFieldName = "STQCCA1" };
  10 customerAddr2 char(25)  { dliFieldName = "STQCCA2" };
  10 customerAddr3 char(25)  { dliFieldName = "STQCCA3" };
end
```

For the complete record layouts in this example database, see *Example DL/I database*.

Such record declarations describe the structure of the database segments as visible to this program. At run time, the runtime PCB determines what an individual program can see.

The structure within a record must match the structure of the segment as DL/I presents it to the program. Define the **keyItem** and **lengthItem** fields with the same length and position that they have in the DL/I segment. If the segment is a logical child, the structure should include the concatenated key of the destination parent as well as the intersection data. If the segment is a concatenated segment in a logical database, the structure should include the concatenated key, the intersection data, and the destination parent segment.

### Database I/O

Because DL/I databases are hierarchical, your program must keep track of the current location within the database. To locate a specific child segment, you must specify the parent segment for that child (if any), the parent segment for that parent (if any), and so on all the way to the root of the database.

Using the customer data base as an example, in order to work with an item record, you must set the customer, location, and order numbers for that item. Fortunately, this is very simple in EGL. Assuming you have correctly defined key fields for all of these records, you could get an item record that you want to update as simply as this:

```
//create instances of the records
myCustomer myCustomerRecordPart;
myLocation myLocationRecordPart;
myOrder    myOrderRecordPart;
myItem     myItemRecordPart;

//build a segment search argument
myCustomer.customerNo = "5001";
myLocation.locationNo = "22";
myOrder.orderDateNo = "20050730A003";
myItem.itemInventoryNo = "CHAIR"; //1st part compound key
myItem.itemLineNo = 27;

//get the item and hold it
try
  get myItem forUpdate;
onException
  myErrorHandler(2);
end
```

At run time, DL/I will perform a GU (get unique) call with a 30-byte concatenated segment search argument that includes the key fields for customer (6 bytes), location (6 bytes), order (12 bytes), and item (6 + 2 bytes).

### Customizing the DL/I statements

Given an EGL statement that uses a DL/I record as the I/O object, you can deal with that statement in one of several ways:

- You can accept the implicit DL/I statement. In this case, changes you make to the DL/I record part affect the DL/I statements that EGL will generate from your code. If you later indicate that you want to use a different record item as the key of a DL/I segment, for example, EGL will automatically pick up that change at generation time.



- You can make the statement explicit, extending the EGL I/O statement with the **with #dli{ *statement* }** syntax. In this case, the details of that DL/I statement are isolated from the DL/I record part, and any subsequent changes made to the DL/I record part have no effect on the DL/I statement that is used at run time. If you remove an explicit DL/I statement from the source, the implicit DL/I statement (if any) is again available at generation time. For more details, see *#dli directive*.
- You can change the name of the PCB by including the **usingPCB** keyword and specifying a substitute PCB name in your EGL I/O statement. With two PCBs you can, for example, keep two different pointers into the same database simultaneously, letting you read a segment twice.
- You can replace the statement entirely and code explicit DL/I calls using the library functions DLILib.AIBTDLI(), DLILib.EGLTDLI(), or VGLib.VGTDLI().

#### Related concepts

"Basic DL/I database concepts"

"Record types and properties" on page 138

"Set-value blocks" on page 68

#### Related tasks

"EGL system exceptions" on page 587

"Displaying implicit DL/I code" on page 325

#### Related reference

"#dli directive" on page 325

"@DLI" on page 322

"add" on page 661

"delete" on page 673

"EGL library DLILib" on page 929

"EGL library VGLib" on page 1047

"Example DL/I database" on page 319

"get" on page 687

"get next" on page 701

"get" on page 687

"IMS and DL/I error codes" on page 361

"PCB record part properties" on page 145

"replace" on page 738

"set" on page 742

## Basic DL/I database concepts

This topic deals with DL/I concepts related to database I/O only. DL/I calls are also used in IMS to handle terminal and program-to-program communications. See *Using non-database PCBs in DL/I programming*.

These terms and concepts are essential in developing a DL/I database program:

#### Segments

The primary unit of data in a DL/I database is the segment. A segment is similar to a record. It is a single block of data divided into data fields.

#### Database Hierarchy

A single database can contain many types of segments. These segments are arranged in a hierarchical (top down) relationship. The segment at the top of the hierarchy is called a root segment. Each segment can have one or more dependent segments related to it at a lower level in the hierarchy. A segment with a dependent segment is called the parent of the dependent segment. The



dependent segment is called a child segment. Each segment in the database, except for a root segment, has one and only one parent. The root segment has no parent.

### **Sequence Field**

Each segment type in a database can have one of its fields designated as a sequence field. The value of the sequence field determines the order in which the segments are stored and retrieved from the database. If a sequence field is defined for the child segment, DL/I stores multiple occurrences of the same child segment in sequence field order under that child's parent.

### **Program Specification Block (PSB)**

A PSB is a formal DL/I description of the hierarchical database structures that a program can access. The EGL record part of type PSBRecord contains the information that your program needs to interact with the DL/I PSB. The PSB shows the hierarchical relationships that exist between types of segments.

### **Program Communication Block (PCB)**

A PCB is an entry in a PSB. Each database PCB describes one hierarchical data structure that a program can use. The data structure might correspond directly to the structure of a physical or logical DL/I database or might invert the database structure through access by a secondary index.

### **DL/I call**

A DL/I call is an invocation of DL/I by a program. The parameter list passed for a database call provides DL/I with the following information:

#### **Function code**

Indicates if DL/I is to get, insert, replace, or delete segments from the database.

#### **Database identifier**

Points to the program communication block (PCB) that identifies the database that DL/I is to access on the call.

#### **I/O area address**

Identifies the address of the buffer that contains the segment after it is read from the database or before it is written to the database.

#### **Segment search argument (SSA) list**

Lists a set of search criteria that enables DL/I to select the segments that it retrieves from the database or specify the position of segments it inserts into the database.

When you code a DL/I program in a language like COBOL or PL/I, you either code the DL/I parameter list directly or use the command level interface of CICS for MVS/ESA™ or CICS for VSE/ESA™ to create the DL/I parameter list. VisualAge Generator creates DL/I parameter lists for you based on the I/O option and the position of the I/O object in the PSB. You can view the DL/I call created for the function. You can also modify the DL/I call to use additional DL/I functions.

### **Database position**

When a program is running, DL/I maintains a position pointer for each PCB in the program PSB. The pointer indicates the place in the database where a SCAN function (DL/I get next function) begins searching for the segment to retrieve.

The position pointer is set on any successful DL/I call to point to the segment following the last segment accessed on the call. If no calls are issued, the

current position indicates the start of the database. If the end of database condition is encountered, the current position becomes the start of the database.

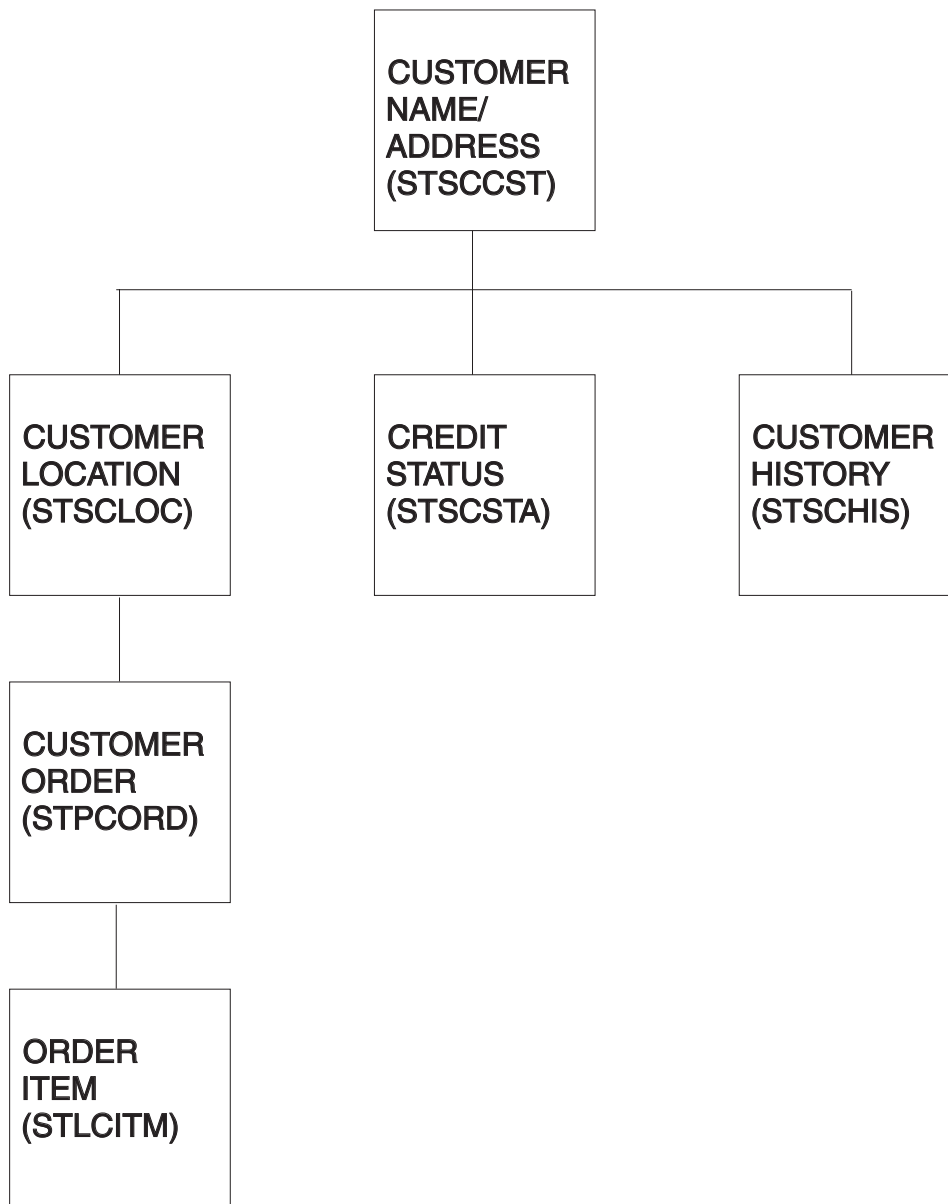
As DL/I continues scanning a database for a segment that satisfies the SSA list criteria, DL/I accesses each root segment in the order it appears in the database. When DL/I finds a root segment, it accesses all the dependents of the root before scanning the next root. As DL/I scans the dependent segments, it first tries to read the next segment at the next lower level. If there is not a lower level, it reads the next segment at the same level. If there are no more segments at the current level, it returns to the previous level to search for the next segment. This process is called the “top to bottom, left to right” search order.

**Related concepts:**

“DL/I database support” on page 310

## **Example DL/I database**

To provide a consistent experience and to let each example build on others, this documentation uses the same example DL/I database wherever possible. This customer database has basic customer information at the root level. For each customer there are segments for credit status, history, and individual locations. Each location has order segments, and each order has line item segments.



EGL represents each of these segments in your program as records of type `DLISegment`. The following code sample shows how you can declare this database structure in EGL. From time to time we will also show examples of DL/I calls using DL/I versions of segment and field names (8 characters maximum). Those DL/I names are also shown in the example.

```

package customer;

//define records to match segments in DL/I db
Record myCustomerRecordPart type DLISegment
{ segmentName="STSCCST", keyItem="customerNo" }
  10 customerNo char(6)      { dliFieldName = "STQCCNO" }; //key field
  10 customerName char(25)   { dliFieldName = "STUCCNM" };
  10 customerAddr1 char(25)  { dliFieldName = "STQCCA1" };
  10 customerAddr2 char(25)  { dliFieldName = "STQCCA2" };
  10 customerAddr3 char(25)  { dliFieldName = "STQCCA3" };
end

Record myLocationRecordPart type DLISegment
{ segmentName="STSCLOC", keyItem="locationNo" }

```

```

10 locationNo char(6)      { dliFieldName = "STQCLNO" }; //key field
10 locationName char(25)   { dliFieldName = "STFCLNM" };
10 locationAddr1 char(25)  { dliFieldName = "STFCLA1" };
10 locationAddr2 char(25)  { dliFieldName = "STFCLA2" };
10 locationAddr3 char(25)  { dliFieldName = "STFCLA3" };
end

Record myOrderRecordPart type DLISegment
{ segmentName="STPCORD", keyItem="orderDateNo" }
10 orderDateNo char(12)    { dliFieldName = "STQCODN" }; //key field
10 orderReference char(25)  { dliFieldName = "STFCORF" };
10 orderItemCount num(6)    { dliFieldName = "STFCOIC" };
10 orderAmount decimal(12,2) { dliFieldName = "STFCOAM" };
end

Record myItemRecordPart type DLISegment
{ segmentName="STLCITM", keyItem="itemKey" }
10 itemKey char(8);        //key field
15 itemInventoryNo char(6)  { dliFieldName = "STKCIIN" };
15 itemLineNo smallint      { dliFieldName = "STQCILI" };
10 itemQtyOrdered num(6)    { dliFieldName = "STFCIQO" };
10 itemQtyShipped num(6)    { dliFieldName = "STFCIQS" };
10 itemQtyBackOrdered num(6) { dliFieldName = "STFCIQB" };
10 itemAmount decimal(12,2) { dliFieldName = "STFCIAM" };
10 itemNumber char(6)       { dliFieldName = "STQIINO" };
10 itemDescription char(25)  { dliFieldName = "STFIIDS" };
10 itemQtyOnHand num(6)      { dliFieldName = "STFIIQH" };
10 itemQtyOnOrder num(6)     { dliFieldName = "STFIIQH" };
10 itemQtyReserved num(6)    { dliFieldName = "STFIIQR" };
10 itemUnitPrice char(6)     { dliFieldName = "STFIIPR" };
10 itemUnitOfIssue char(1)   { dliFieldName = "STFIIUN" };
end

Record myCreditRecordPart type DLISegment
{ segmentName="STSCSTA" }
10 creditLimit char(12)     { dliFieldName = "STFCSSL" };
10 creditBalance char(12)   { dliFieldName = "STFCSBL" };
end

Record myHistoryRecordPart type DLISegment
{ segmentName="STSCHIS", lengthItem="historySegmentLength", keyItem="historyDateNo" }
10 historySegmentLength smallint { dliFieldName = "STGCSL" };
10 historyDateNo char(12)        { dliFieldName = "STQCHDN" };
10 historyReference char(25)      { dliFieldName = "STFCHRF" };
10 historyItemCount smallint      { dliFieldName = "STFCHIC" };
10 historyAmount decimal(12,2)    { dliFieldName = "STQCHAM" };
10 historyStatus char(77)         { dliFieldName = "STQCLOS" };
end

//declare overall db layout in PSB
Record myCustomerPSBRecordPart type PSBRecord { defaultPSBName="STBICLG" }
// three PCBs required for CBLTDLI on IMS
iopcb IO_PCBRecord { @PCB { pcbType = TP } };
elaalt ALT_PCBRecord { @PCB { pcbType = TP } };
elaexp ALT_PCBRecord { @PCB { pcbType = TP } };

// database PCB
customerPCB DB_PCBRecord { @PCB {
pcbType = DB,
pcbName = "STDCDBL",
hierarchy = [
@Relationship { segmentRecord = "myCustomerRecordPart" },
@Relationship {
segmentRecord = "myLocationRecordPart", parentRecord = "myCustomerRecordPart" },
@Relationship {
segmentRecord = "myOrderRecordPart", parentRecord = "myLocationRecordPart" },
@Relationship {

```

```

        segmentRecord = "myItemRecordPart", parentRecord = "myOrderRecordPart" },
    @Relationship {
        segmentRecord = "myCreditRecordPart", parentRecord = "myCustomerRecordPart" },
    @Relationship {
        segmentRecord = "myHistoryRecordPart", parentRecord = "myCustomerRecordPart" }}}};
end

program PrintCatalog type basicProgram {
    @DLI{
        psb = "myPSB",
        callInterface = CBLTDLI,
        // leave space for IO and ALT PCBs
        pcbParms = ["", "", "", "customerPCB"] } }

    //create instances of the records
    myCustomer myCustomerRecordPart;
    myLocation myLocationRecordPart;
    myOrder myOrderRecordPart;
    myItem myItemRecordPart;
    myCrStatus myCreditRecordPart
    myHistory myHistoryRecordPart

    myPSB CustomerPSB;

    function main()
    ...
end
end

```

#### Related concepts:

“Basic DL/I database concepts” on page 317

“DL/I database support” on page 310

“Record types and properties” on page 138

#### Related reference:

“@DLI”

## @DLI

The program property **@DLI** allows you to specify behaviors for your program’s DL/I calls by means of a set-value block. When you include this property in a program, EGL can access details about your most recent call to a DL/I database. For more information, see *DLIVar*.

The **@DLI** property contains the following fields:

#### callInterface DLICallInterfaceKind

The **callInterface** property defines aspects of the IMS and DL/I calls that the EGL-generated COBOL program implements in response to EGL statements like **add** and **get**. It is recommended that you use AIBTDLI, although use of that setting requires your organization to configure PSBs as appropriate for the AIBTDLI interface.

The values are as follows:

#### AIBTDLI (the default)

This more recent interface takes advantage of the IMS Application Interface Block (AIB). The AIB allows you to access runtime PCBs by name rather than by address. Before you use this interface, however, make sure that the system programmer in your organization has assigned a symbolic name to each of the runtime PCBs; specifically, the programmer must set the PCBNAME parameter in the PSBGEN definition.

You identify the name by setting the `PCBName` property of the PCB record. The default for that property is the name of the record.

### **CBLTDLI**

Access to a PCB is slightly faster because your program uses an address rather than a PCB name. However, requirements for defining the PSBRecord parts are greater than with AIBTDLI, and more important, you usually must pass a PSB record or PCB records to called programs rather than simply relying on a name in a given PCB record in the called program. Aside from handling extra data in a program call, the passing of a PSB record requires that you set the called program's **psbParm** property, and the passing of PCB records requires that you set the called program's **pcbParms** property. (If you pass both a PSB record and PCB records, the PSB is ignored.)

### **psb STRING**

The **psb** property identifies the PSB record that refers to the Program Specification Block (runtime PSB) to be scheduled with the program. The record must be in scope.

In environments other than CICS, you cannot change the runtime PSB that is the first one scheduled in the run unit.

### **pcbParms STRING[]**

As used in a called program that receives PCB records as parameters, the property field **pcbParms** provides a list of strings that allow EGL to match each parameter (of a PCB record type) with a PCB record in the program PSB record part. The property has no effect if the value of the **callInterface** property is AIBTDLI.

As shown in the next example, the position of strings in the array must match the position of PCB records in the program PSB record part, and each non-empty string in the array must be identical to the name of a PCB record in the list of program parameters.

```
Record myPSBRecordPart type PSBRecord {defaultPSBName = "ibmPSB"}
```

```
    // details of the following records are omitted in this example
    ioPCB IO_PCBRecord;
    dbPCB DB_PCBRecord;          // passed in
    db2PCB DB_PCBRecord;
    gsamPCB GSAM_PCBRecord;      // passed in
    gsam2PCB GSAM_PCBRecord;
end
```

```
program PrintCatalog type basicProgram
(GSAM_PCB_parm GSAM_PCBRecord, DB_PCB_parm DB_PCBRecord) {
  @DLI{
    psb = "myPSB",
    callInterface = CBLTDLI,
    pcbParms = ["", "DB_PCB_parm", "", "GSAM_PCB_parm", ""]
  }
}
```

```
myPSB myPSBRecordPart;
```

If you specify properties **pcbParms** and **psbParm**, the PCB-specific addresses in the former override the equivalent addresses in the latter.

Although an empty string is used for each PCB record that is in the PSB record part but is not matched by a parameter, you can avoid specifying the last

elements in the array if *those* elements refer to PCB records that are not matched by a parameter. The following assignment is also valid in the current example:

```
pcbParms = ["", "DB_PCB_parm", "", "GSAM_PCB_parm"]
```

To avoid an error in array-element positioning if you later add a PCB record to the PSB record part, include (without exception) an array element for each PCB record in the PSB record part.

If a PCB record in the PSB record part is a redefine of another PCB record in that part, the original record and the redefined record represents the same area of memory and are counted only once as you construct the array for **pcbParms**.

#### **psbParm STRING**

If non-EGL program calls an EGL program and includes a 12-byte name and address for accessing a runtime PSB, the called program property **psbParm** identifies the parameter that provides the name and address.

The parameter is of type PSBDataRecord, which is structured as follows:

```
Record PSBDataRecord
  psbName char(8);
  psbRef int;
end
```

Here is an example of the property in use:

```
Program Prog1 ( psbData PSBDataRecord )
{
  @DLI (psbParm = "psbData" )
}
```

If you specify properties **pcbParms** and **psbParm**, the PCB-specific addresses in the former override the equivalent addresses in the latter.

The name and address received into the program are assigned automatically to the system variable **DLILib.psbData**.

#### **handleHardDLIErrors BOOLEAN**

Sets the default value for the system variable **VGVar.handleHardDLIErrors**.

The variable controls whether a program continues to run after a hard error has occurred on a DL/I or IMS I/O operation in a **try** block. The default value for the property is *yes*, which sets the variable to 1.

Code that was migrated from VisualAge Generator may not work as before unless you set **handleHardDLIErrors** to *no*, which sets the variable to 0.

For details, see *DLIVar* and *Exception handling*.

#### **Related concepts**

"DL/I database support" on page 310

#### **Related reference**

"DLIVar" on page 1062

"Exception handling" on page 94

"PCB record part properties" on page 145

## #dli directive

The **#dli** directive lets you modify the default DL/I calls that EGL generates from I/O keywords. Place the directive in the same line as the EGL code it modifies, as in the following example, and continue onto extra lines as necessary:

```
get myLocation with #dli{
  GU STSCCST (STQCCNO = :myCustomer.customerNo)
  STSCLOC (STQCLNO = :myLocation.locationNo) };
```

As you may have noticed, the syntax in the above example is not exactly the same as in DL/I. EGL supports a powerful pseudo-DL/I syntax that takes on some of the burden of DL/I formatting for you. For example, EGL converts all names to upper case, adds spaces to names to bring them to eight characters where necessary, and converts EGL relational operators such as "!=" to DL/I equivalents. In addition, EGL allows you to use host variables (variables defined in the host EGL program and not in the DL/I database) in your pseudo-DL/I calls. In the sample code above, the host variables begin with a colon (:). EGL turns all these raw materials into properly formatted DL/I calls at generation time.

Occasions when you might choose to use the **#dli** directive include the following:

- You need to add a DL/I command code to extend the function of the call. Examples include using a concatenated key (\*C), or excluding a segment in a path call from replacement (\*N).
- You are making a **get** or **get next** call using a dynamic array as your operand. Filling a dynamic array with a **get** involves an initial GU call followed by a loop of GN calls until the array is full or DL/I runs out of segment occurrences. Because there is no key field for the array itself (only for members of the array), the default DL/I code that EGL generates will not work. You will need to create code something like the following:

```
myOrderArray myOrderRecordPart[] {maxsize = 20}; //array of orders
```

```
get myOrderArray with #dli{
  GU STSCCST STSCLOC STPCORD
  GN STPCORD };
```

```
//get the next 20 orders
get next myOrderArray;
```

- You wish to retrieve a segment based on a field other than the key, or on a value not in the segment itself.
- You are reading all segments in the database, regardless of segment type. This requires a GN with no SSA.

You will find specific instructions for a number of these tasks in *DL/I-specific tasks*.

### Related concepts:

"DL/I database support" on page 310

### Related reference

"get" on page 687

"get next" on page 701

## Displaying implicit DL/I code

EGL generates a default DL/I call from EGL I/O keywords. You can display this DL/I call to allow you to edit it directly:

1. Open the EGL source file that contains the EGL I/O statement. If you do not have the file open, right-click on the EGL source file in the Project Explorer, then select **Open With > EGL Editor**.



2. Click on the EGL I/O statement, then right-click. A context menu displays.
3. Click **DLI Statement > View**. The View DLI Statement window opens, showing the default DL/I call for this EGL I/O statement.
4. In the View DLI Statement window, click **Add Statement**. The default DL/I call is added to the EGL I/O statement.

To add the default DL/I call without previewing it in the View DLI Statement window, right-click the EGL I/O statement and click **DLI Statement > Add**.

#### Related concepts

"DL/I database support" on page 310

"Basic DL/I database concepts" on page 317

#### Related reference

"#dli directive" on page 325

## DL/I-specific tasks

Here are specific instructions for performing the following DL/I tasks:

- "Using path calls to access multiple segments"
- "Reading all segments with a single call" on page 327
- "Searching with a secondary index" on page 327
- "Searching with another non-key field" on page 329

All the examples in these instructions use the *Example DL/I database*.

### Using path calls to access multiple segments

If you invoke an EGL I/O statement with a dependent segment record object, you can read in any of the segments on the path from the root to the object at the same time. You can do this simply by adding the records for the path call to the statement. For example, when retrieving an order segment from our example customer database, you can read in the customer segment and the location segment on the same call:

```
get myCustomer, myLocation;
```

This statement generates the following DL/I pseudocode:

```
GU STSCCST*D (STQCCNO = :myCustomer.customerNo)
  STSCLOC (STQCLNO = :myLocation.locationNo)
```

If you use the D command code on a **get...forUpdate** statement, the subsequent **replace** statement affects every segment retrieved. You can prevent replacement of a selected segment by specifying an explicit N command code in the SSA for the **replace** keyword, as in the following example:

```
get myCustomer, myLocation forUpdate;
replace myLocation with #dli{
  REPL STSCCST*N
  STSCLOC };
```

The default DL/I call EGL builds for a **delete** function that follows a **get forUpdate** statement with D command codes does not delete each segment retrieved. It deletes only the I/O object segment.

## Reading all segments with a single call

You can use a single function to read all segments in a database. If you issue a DL/I **get next** call with no SSAs, DL/I returns the next segment in the database regardless of its type. Follow these steps to use this technique:

1. Write a **get next** statement for the record that represents the largest segment in the database. This ensures that any segment you read will not exceed allocated memory.
2. Edit the statement's default DL/I call to delete the single SSA.
3. Create records matching the other segments in the database. Declare them as redefined records for the record in step 1.
4. Check `DLIVar.segmentName` after the **get next** statement to determine the type of segment that was retrieved.
5. Access the retrieved segment from the redefined record structure, or assign the redefined structure to another record of the same type.

Here is an example of code that will print everything in the customer database. In this example, `myHistoryRecordPart` is the largest `DLISegment` record:

```
redefCustomer myCustomerRecordPart {redefines=myHistoryRecordPart};
redefLocation myLocationRecordPart {redefines=myHistoryRecordPart};
...
```

```
//read next segment, whatever type it is, into history record
while (myHistory not EOF)
  get next myHistory with #dli{
    GN };

//so what type was it?
case (DLIVar.segmentName)
  when "STSCCST"           // it was a customer
    myCustomer = redefCustomer;
    printCustomer();        // myCustomer is global
  when "STSCLOC"           // it was a location
    myLocation = redefLocation;
    printLocation();
  ...
end
end
```

## Searching with a secondary index

There are several ways to add secondary indexing to a database. The simplest is to add a **secondaryIndex** field to the `@PCB` property to the `DB_PCBRecord` variable in your PSB record. If this is the first `DB_PCBRecord` in the PSB record, EGL will use this secondary index by default. For example, if you wanted to search for customers based on the `orderReference` field (`STFCORF`) in the orders segment, you could add a secondary index to the `customerPCB` in our example database, as follows:

```
// database PCB
customerPCB DB_PCBRecord { @PCB {
  pcbType = DB,
  pcbName = "STDCDBL",
  secondaryIndex = "STFCORF", //use DL/I name
  hierarchy = [
    @Relationship { segmentRecord = "myCustomerRecordPart"},
    @Relationship {
      segmentRecord = "LocationRecord",
      parentRecord = "myCustomerRecordPart" },
  ]
}
```

```

@Relationship {
    segmentRecord = "OrderRecord",
    parentRecord = "myLocationRecordPart" },
...

```

Perform a **get** operation to locate a customer, as in the following example:

```
get myCustomer;
```

EGL will generate the following code by default:

```
GU STSCCST (STFCORF = :myOrder.orderReference)
```

If you wish to be able to choose between accessing customers by orderReference or by customerNo, create a second PCB for your secondary index. In the following example the second PCB is named orderReferencePCB:

```

// database PCB--access by customer number
customerPCB DB_PCBRecord { @PCB {
    pcbType = DB,
    pcbName = "STDCDBL",
    hierarchy = [
        @Relationship { segmentRecord = "myCustomerRecordPart"},
        @Relationship {
            segmentRecord = "LocationRecord",
            parentRecord = "myCustomerRecordPart" },
        @Relationship {
            segmentRecord = "OrderRecord",
            parentRecord = "myLocationRecordPart" },
        ...
    ]
}

// database PCB--access by order reference
orderReferencePCB DB_PCBRecord { @PCB {
    pcbType = DB,
    pcbName = "STDxDBL",
    secondaryIndex = "STFCORF", //use DL/I name
    hierarchy = [
        @Relationship { segmentRecord = "myCustomerRecordPart"},
        @Relationship {
            segmentRecord = "LocationRecord",
            parentRecord = "myCustomerRecordPart" },
        @Relationship {
            segmentRecord = "OrderRecord",
            parentRecord = "myLocationRecordPart" },
        ...
    ]
}

```

The pcbName must match an actual DL/I PCB. Now EGL's default behavior will once more be to access a customer record using the customerNo field. To access it using the alternate key, your EGL I/O statement must specify the orderReferencePCB with the **usingPCB** keyword, as in the following example:

```
get myCustomer usingPCB orderReferencePCB;
```

You can also have a more complex case, where you want to change the structure of the entire database as your program sees it. (Again, keep in mind that the PCB structure in your EGL program must match an existing DL/I PCB.) Suppose you want the customer database to look like an orders database to your program, with the unique reference number as the key to the orders segment. You can have a PCB with the following structure:

```

// orders view of customer database
ordersPCB DB_PCBRecord { @PCB {
    pcbType = DB,
    pcbName = "STDCDBL",
    secondaryIndex = "STFCORF", //use DL/I name
    hierarchy = [

```

```

@Relationship { segmentRecord = "myOrderRecordPart" },
@Relationship {
    segmentRecord = "myLocationRecordPart",
    parentRecord = "myOrderRecordPart" },
@Relationship {
    segmentRecord = "myCustomerRecordPart",
    parentRecord = "myLocationRecordPart" },
@Relationship {
    segmentRecord = "myCreditRecordPart",
    parentRecord = "myCustomerRecordPart" },
@Relationship {
    segmentRecord = "myHistoryRecordPart",
    parentRecord = "myCustomerRecordPart" },
@Relationship {
    segmentRecord = "myItemRecordPart",
    parentRecord = "myOrderRecordPart" }]]];
end

```

Assuming the order reference number is unique to each customer and order, and assuming ordersPCB is now your default PCB, you can find the customer for an order by doing a modified path call that removes the qualifications for the location and the customer:

```

get myOrder, my Customer with #dli{
    GU STPCORD (STQCODN = :myOrder.orderReference)
    STSCLOC
    STSCCST };

```

## Searching with another non-key field

You can use any field in a segment as a search argument on a DL/I call by modifying the search arguments (SSAs) for the call. For example, if you wanted to read through the customer database and retrieve the customer segment and credit segment for each customer with a credit balance greater than a specified amount, you would define the DL/I call search arguments as follows:

1. You want to search on the creditBalance field (STFCSBL) in the credit segment (STSCSTA). To do this, define a variable of type MONEY (for example, "targetBalance") that contains the specified amount that you want to search for.
2. Write a **get** statement for the myCrStatus record.
3. Add a #dli directive to the line, modifying the default code. Add a qualified SSA that looks for a segment where the amount in the creditBalance field is greater than targetBalance.
4. Include a path command code (\*D) to retrieve the customer segment (STSCCST) that corresponds to the credit segment.

The following sample code illustrates this process:

```

targetBalance MONEY;
targetBalance = 10,000.00;

get myCrStatus with #dli{
    GU STSCCST*D STSCSTA (STFCSBL >= :targetBalance) };

```

You might also want to search based on information in another record. For example, if you wish to look up a customer based on a customer number (invCustNo) in an invoice record (myInvoice) that is of a type (myInvoiceRecordPart) that is a basic record and not part of the database. The code would look something like the following:

```

get myCustomer with #dli{
    GU STSCCST (STQCCNO = :myInvoice.invCustNo) };

```

**Related concepts:**

"DL/I database support" on page 310

"TMS runtime support" on page 345

**Related reference:**

"DLIVar" on page 1062

"Example DL/I database" on page 319

## DL/I considerations for CICS environments

In the CICS environment, PSB scheduling is handled differently than in non-CICS environments. The following sections describe PSB scheduling, how to use an alternate PSB at run time and how to share a PSB with a called program.

### Understanding PSB scheduling

A CICS DL/I program must schedule a PSB before the program can access databases defined in that PSB. The program must end the PSB when database access is complete. Enterprise Developer Server for z/OS automatically handles PSB scheduling for a program. However, you need to know when the PSB is scheduled because the following functions are affected by PSB scheduling:

#### Segment record locking

Segment update locks are released when the PSB is ended.

#### Database positioning

Database position is lost when the PSB is ended.

#### Update commitment

Changes are committed (written to the database) when the PSB is ended.

The PSB is scheduled whenever a DL/I call is issued for the program and the PSB is not currently scheduled. The PSB named in `dliLib.psbData.psbName` is the PSB scheduled.

When using a CALL or DXFR statement to transfer program control, the transferred-to program does not have to use the same PSB the transferring program uses.

The PSB ends whenever a CICS SYNCPOINT or SYNCPOINT ROLLBACK is issued. SYNCPOINTS occur when one of the following occurs:

- The top-level program in a run unit ends successfully and returns control to CICS. For CICS, a run unit is equivalent to a single transaction and consists of all EGL programs and non-EGL programs that transfer control among themselves using a DXFR or CALL statement. For non-EGL programs, this also includes any transfer that uses a CALL statement, CICS LINK command, or CICS XCTL command.
- A program uses a CONVERSE I/O option, and either of the following is set to 1:
  - `converseVar.segmentedMode` (defaults to 1 if the program is defined as segmented)
  - `converseVar.commitOnConverse`
- A transfer using an XFER statement occurs
- A program calls either the `sysLib.commit()` function or the COMMIT service.
- A transfer using a DXFR statement occurs, a PSB is scheduled, and one of the following occurred:
  - Transfer to a non-EGL program when a PSB is scheduled.

- The synchOnPgmTransfer generation option was set to yes for the transferred-from program
- The synchOnPgmTransfer generation option was set to no for the transferred-from program and different PSB names were identified in the program specifications for the two programs.
- An EGL-called DL/I program returns to the calling non-EGL program, the PSB was not passed in the DLILib.psbData structure, and PCBs were not passed using PCB records.

A SYNCPOINT ROLLBACK occurs when:

- An EGL program calls the sysLib.rollback() function or the RESET service.
- A program ends because of an error condition.

When a rollback occurs, all changes that were made to databases and recoverable files since the start of the logical unit of work (LUW) are backed out.

### Using an alternate PSB at run time

EGL uses the PSB named in the program specification to create DL/I calls for the segments defined in the PSB and to validate any changes you make to the DL/I calls. In the CICS environment, Enterprise Developer Server for z/OS also uses the PSB name for PSB scheduling at the time the program is run.

Sometimes, however, you might want to use an alternate PSB with the same program. The alternate PSB describes databases with the exact same structure as the program PSB, but the databases themselves might be different. For example, you could have a set of test databases for program development and a corresponding set of production databases that contain the real data for production.

If you want to use an alternate PSB for any reason, your program can dynamically change the PSB that is scheduled by moving the alternate PSB name into DLIVar.dliPsbName before running the first DL/I function. The alternate PSB must match the program PSB except that the database names can be different.

### Sharing a scheduled PSB with a called program

Called and calling programs cannot both be DL/I programs unless they share the same program PSB or unless a commit is done to end the PSB prior to each call or return to a program that uses a different PSB. The PSB is shared by passing the DLILib.psbData structure or the PCB records on the call for both the called and the calling program.

If the called and calling programs are both EGL programs, and if the PSB was scheduled before the call, EGL does not reschedule the PSB in the called program.

You can share a PSB between EGL programs and non-EGL programs. When the DLILib.psbData structure is passed as a parameter, a 12-byte area is actually passed. The first 8 bytes contain the PSB name; the final 4 bytes contain the address of the CICS User Interface Block (UIB). If the PSB is not scheduled, the UIB address is 0. When sharing a PSB between EGL programs and non-EGL programs, the called program should check the UIB address before scheduling. If the UIB address is not 0, the PSB should not be rescheduled. If the called program ends the PSB, it must set the UIB address field to 0. If the PSB is scheduled again, the UIB address field should be set to the UIB address returned by CICS.

If a program needs to share a scheduled PSB with a called EGL program, it should pass a 12-byte area to the program. Again, the first 8 bytes should contain the PSB name and the next 4 bytes should contain the UIB address. If the PSB is not scheduled, the UIB address should be 0. On return, the UIB address reflects the current scheduling status of the PSB.

### **Recovering after a deadlock in record queuing**

Updated records are not actually written to the database until the PSB is ended. Your program obtains exclusive use of these records until PSB termination because the EGL **get...forUpdate** I/O statement locks out other programs from changing the record again until it is actually written to the database. This can result in a deadlock situation where your program has some record locked and now wants to change records locked by another program that in turn needs the records you have locked. When CICS detects a deadlock situation, it abnormally ends one of the programs, backs out the changes it has made to the database, and writes an error message to the terminal explaining why the program ended.

If having a program abnormally end is unacceptable to the users of the program, you can define your program as restartable in the CICS tables. For information on restarting programs, see "Restarting EGL programs after a DL/I deadlock" later in this topic.

If the program is restartable and CICS detects a deadlock situation, CICS backs out the changes that the program has made since the PSB was scheduled and restarts the program from the beginning of the transaction.

If your program is running in segmented (pseudoconversational) mode, the most recent segment of the program is restarted, and you do not need any special restart code in the program.

If a conversational program is restarted, the program begins again at the top. You can determine that the program was restarted by having the program test for a value of 1 in the DLIVar.cicsRestart field. If DLIVar.cicsRestart is 1, you can write a message to the program user explaining that the program was restarted because of a deadlock and then display the initial program map again.

### **Restarting VisualAge Generator programs after a DL/I deadlock**

When the DL/I program isolation facility is used, deadlocks can occur between two transactions locking on the same record. The programs try to update the same record at the same time. If both updates are accepted, one of the changes is lost. If CICS for MVS/ESA or CICS for VSE/ESA detects that data is lost, it abnormally ends the transaction with an ADLD abend code.

The Enterprise Developer Server for z/OS abend handler requests restart for programs that end with a deadlock abend. CICS for MVS/ESA or CICS for VSE/ESA restarts the transaction from the beginning if:

- You have specified DTB=YES and RESTART=YES in the PCT for the transaction program.
- The CICS for MVS/ESA or CICS for VSE/ESA transaction restart program (DFHRTY) specifies to restart the transaction.
- The temporary storage queues are defined as recoverable.

Otherwise, CICS for MVS/ESA or CICS for VSE/ESA writes a message indicating the reason for program termination. A restarted program is a program that is



started again from the beginning of the last transaction that was running. All changes to databases that were made since the program PSB was last scheduled are rolled back.

The distributed version of the CICS for MVS/ESA or CICS for VSE/ESA restart program (DFHRTY) does not restart an EGL-generated program from the beginning of the last database transaction that was running. You can add code to DFHRTY to restart EGL database transactions. Your program should check that the following is true:

- The current abend code is ADLD.
- The transaction identifier is the identifier of the transaction you want restarted.
- The restart count is less than the specified number. This is a restart loop check.

If all checks are met, your program should set the restart flag to on, indicating to CICS for MVS/ESA or CICS for VSE/ESA that restart is to continue. For more information on modifying DFHRTY, refer to the recovery, restart, and customization manuals for your CICS for MVS/ESA or CICS for VSE/ESA system.

Programs running in segmented mode must have all the CICS for MVS/ESA or CICS for VSE/ESA resources (that are to be updated) defined as recoverable if restart is requested. The names of the temporary storage queues that are created by the segmentation function of Enterprise Developer Server for z/OS are a combination of the user's terminal identifier appended to a 4-character prefix. The prefixes used are in the form of X'EE' followed by either WRK or MSG.

You should not request restart for conversational programs unless you have designed the program to handle restart. A program can test the `DLIVar.cicsRestart` field to see if the current program transaction has been restarted. One simple way of designing a program for restart is to have the program test `cicsRestart` whenever it begins. If the restart flag is on, a special message or map should appear to user 1 explaining that the transaction was restarted at the beginning because user 2 was changing the database at the same time. User 1's changes were backed out, and the program was restarted to prevent the changes from being lost.

### Accessing distributed DL/I databases

Programs running on CICS for MVS/ESA, CICS for VSE/ESA, or CICS for OS/2® systems can access DL/I databases on remote systems by calling a CICS for MVS/ESA- or CICS for VSE/ESA-called batch server program that runs on the system where the database is located.

## DL/I considerations for non-CICS environments

In non-CICS environments, PSB scheduling is handled differently than in CICS environments. The following sections describe PSB scheduling and the use of an alternate PSB at run time.

### Understanding PSB scheduling

During program execution for IMS/VS, IMS BMP, and z/OS batch, DL/I initialization schedules a single PSB. This PSB is the only one available for one batch job step or IMS transaction.

All EGL programs and non-EGL programs in the run unit must share the same PSB. The run unit includes all programs that you call or transfer to using a **transfer** statement of the form *transfer to a program*. For IMS BMP and z/OS batch, the run unit also includes all programs that you transfer to using a **transfer** statement of the form *transfer to a transaction*. For IMS/VS, the run unit for a



program that you transfer to using a **transfer** statement of the form *transfer to a transaction* differs from that of the transferring program.

For IMS/VS, you specify the PSB that is to be used in the IMS system definition. The IMS PSB must have the same name as the program name. The PSB is scheduled at the beginning of the IMS transaction.

For IMS BMP and z/OS batch, you specify the name of the PSB to be used in the JCL used to run the batch job. The PSB is scheduled at the beginning of the IMS BMP or z/OS batch job.

## Understanding commit points and the logical unit of work

A logical unit of work (LUW) ends whenever a commit point or a rollback occurs.

A commit point occurs when one or more of the following events happen:

- The top-level program in a run unit ends successfully.  
For z/OS batch and IMS BMP, a run unit consists of all EGL programs and non-EGL programs that transfer control among themselves using a **transfer** or **call** statement. For non-EGL programs, this also includes any transfer that uses an OS XCTL macro or a **call** statement.  
For IMS/VS, a run unit is equivalent to a single transaction and consists of all EGL programs and non-EGL programs that transfer control among themselves using a transfer statement of the form *transfer to a program* or **call** statement. For non-EGL programs, this also includes any transfer that uses a **call** statement.
- A program uses a **converse** I/O statement, and either of the following is set to 1:
  - converseVar.segmentedMode (defaults to 1 if the program is defined as segmented)
  - converseVar.commitOnConverse

The best time for a commit point to occur is after terminal output and before the next terminal input. A commit point at terminal I/O synchronizes updates to the database and confirmation messages to the program user.

- For z/OS batch and batch-oriented IMS BMP programs, a program transfers using a **transfer** statement of the form *transfer to a transaction* and the synchOnTrxTransfer generation option is set to YES for the program from which you are transferring
- A program calls the sysLib.commit() function or the COMMIT service.  
For z/OS batch, EGL programs that do not use DL/I issue a commit point only if the program has made changes to an SQL table. A commit point does not occur for changes to an SQL table made by a non-EGL program.  
For IMS/VS and transaction-oriented IMS BMP programs (programs that scan a serial file associated with the I/O PCB), sysLib.commit() is ignored. A commit point occurs whenever there is a GU (get unique) call to the I/O PCB.
- For IMS/VS and transaction-oriented IMS BMP programs, a program does a successful get unique to the I/O PCB.

A rollback occurs when one or more of the following events happen:

- An EGL program calls the sysLib.rollback() function
- A program ends because of an error condition

When a rollback occurs, all changes that were made to databases and recoverable files since the start of the LUW are backed out. Rollback does not affect DL/I databases in the VSE batch environment.

## Using an alternate PSB at run time

The actual DL/I PSB name you use at run time might differ from the name you specified as the PSB name during program specification. However, for IMS/VS, IMS BMP, and z/OS batch, the PCB number, type, and order must match in the IMS PSB and the VisualAge Generator PSB. Although the database structures must match, the database names do not have to match the name specified in your EGL PSB definition.

## Using symbolic checkpoint and restart functions (z/OS Batch and IMS BMP Only)

When you run a batch program, you can use `sysLib.commit()` to periodically commit database updates. Alternatively, you can use `EGLTDLI()` to implement symbolic checkpoint and restart functions. Both `sysLib.commit()` and the symbolic checkpoint function commit database updates. However, the symbolic checkpoint function also enables you to save information, such as control totals or the key of the last database record that was processed when a commit point occurred. If the program does not complete successfully, it is backed out to the last commit point. If you have saved information using symbolic checkpoint, when you restart the program, you can restore the saved data using the DL/I restart (XRST) call. You can use this information to resume processing at the point in the database where processing stopped.

### Related concepts:

“DL/I database support” on page 310

### Related reference:

“call” on page 665

“converse” on page 672

“DL/I considerations for CICS environments” on page 330

“EGLTDLI()” on page 930

“transfer” on page 752

---

## VSAM support

VSAM support for EGL-generated Java code is as follows:

- AIX-based code can access local VSAM files
- The following code can access remote VSAM files on z/OS:
  - EGL-generated Java code that runs on Windows 2000/NT/XP
  - The EGL debugger, which runs on Windows 2000/NT/XP

## Access prerequisites

Access requires that you first define the VSAM file on the system where you want the file to reside. Remote access from Windows 2000/NT/XP (whether for the EGL debugger or at run time) also requires that you install Distributed File Manager (DFM) on the workstation as follows:

1. Locate the following file in your EGL installation directory:  
`workbench\bin\VSAMWIN.zip`
2. Unzip the file into a new directory and follow the directions in the `INSTALL.README` file.

## System name

To access a local VSAM file, specify the system name in the resource associations part and use the naming convention that is appropriate to the operating system. To

access a remote VSAM file from the EGL debugger or from EGL-generated Java code, specify the system name in the following way:

```
\machineName\qualifier.fileName
```

*machineName*

The SNA LU alias name as specified in the SNA configuration

*qualifier.fileName*

The VSAM data set name, including a qualifier

The naming convention is similar to the Universal Naming Convention (UNC) format. For details on UNC format, refer to the *Distributed FileManager User's Guide*, which is in the following file in your EGL installation directory:

```
workbench\bin\VSAMWIN.zip
```

---

## MQSeries support

EGL supports access of MQSeries message queues on any of the target platforms. You can provide such access in either of the following ways:

- Use MQSeries-related EGL keywords like **add** and **get next** on an MQ record; in this case, EGL hides details of MQSeries so you can focus on the business problem your code is addressing
- Invoke EGL functions that call MQSeries commands directly, in which case some commands are available that are not supported by the EGL keywords

You can mix the two approaches in a given program. For most purposes, however, you use one or the other approach exclusively.

Regardless of your approach, you can control various runtime conditions by customizing *options records*, which are global basic records that EGL runtime services passes on calls to MQSeries. When you declare an options record as a program variable, you can use an EGL-installed options record part as a typedef; or you can copy the installed part into your own EGL source file, customize the part, and use the customized part as a typedef.

Your approach determines how EGL runtime services makes the options records available to MQSeries:

- If you are working with the EGL **add** and **get next** statements, you identify the options records when you specify properties of an MQ record. If you do not identify a particular options record, EGL uses a default.
- If you are invoking the EGL functions that call MQSeries directly, you use options records as arguments when you invoke the functions. Defaults are not available in this case.

For details on options records and on the values that are passed to MQSeries by default, see *Options records for MQ records*. For details on MQSeries itself, refer to these documents:

- *An Introduction to Messaging and Queueing* (GC33-0805-01)
- *MQSeries MQI Technical Reference* (SC33-0850)
- *MQSeries Application Programming Guide* (SC33-0807-10)
- *MQSeries Application Programming Reference* (SC33-1673-06)

## Connections

You connect to a queue manager (called the *connecting queue manager*) the first time you invoke a statement from the following list:

- An EGL **add** or **get next** statement that accesses a message queue
- An invocation of the EGL function MQCONN or MQCONNX

You can access only one connecting queue manager at a time; however, you can access multiple queues that are under the control of the connecting queue manager. If you wish to connect directly to a queue manager other than the current connecting queue manager, you must disconnect from the first by invoking MQDISC, then connect to the second queue manager by invoking **add**, **get next**, MQCONN, or MQCONNX.

You can also access queues that are under the control of a *remote queue manager*, which is a queue manager with which the connecting queue manager can interact. Access between the two queue managers is possible only if MQSeries itself is configured to allow for that access.

Access to the connecting queue manager is terminated when you invoke MQDISC or when your code ends.

## Include message in transaction

You can embed queue-access statements in a unit of work so that all your changes to the queues are committed or rolled back at a single processing point. If a statement is in a unit of work, the following is true:

- An EGL **get next** statement (or an EGL MQGET invocation) removes a message only when a commit occurs
- The message placed on a queue by an EGL **add** statement (or an EGL MQPUT invocation) is visible outside the unit of work only when a commit occurs

When queue-access statements are not in a unit of work, each change to a message queue is committed immediately.

An MQSeries-related EGL **add** or **get next** statement is embedded in a unit of work if the property **includeMsgInTransaction** is in effect for the MQ record. The generated code includes these options:

- For MQGET, MQGMO\_SYNCPOINT
- For MQPUT, MQPMO\_SYNCPOINT

If you do not specify the property **includeMsgInTransaction** for an MQ record, the queue-access statements run outside of a unit of work. The generated code includes these options:

- For MQGET, MQGMO\_NO\_SYNCPOINT
- For MQPUT, MQPMO\_NO\_SYNCPOINT

When your code ends a unit of work, EGL commits or rolls back *all* recoverable resources being accessed by your program, including databases, message queues, and recoverable files. This outcome occurs whether you use the system functions (**sysLib.commit**, **sysLib.rollback**) or the EGL calls to MQSeries (MQCMIT, MQBACK); the appropriate EGL system function is invoked in either case.

A rollback occurs if an EGL program terminates early because of an error detected by EGL runtime services.

## Customization

If you wish to customize your interaction with MQSeries rather than relying on the default processing of **add** and **get next** statements, you need to review the information in this section.

### EGL dataTable part

A set of EGL dataTable parts is available to help you interact with MQSeries. Each part allows EGL-supplied functions to retrieve values from memory-based lists at run time. The next section includes details on how data tables are deployed.

### Making customization possible

To make customization possible, you must bring a variety of installed EGL files into your project *without changing them in any way*. The files are as follows:

#### records.egl

Contains basic record parts that can be used as typedefs for the options records that are used in your program; also includes structure parts that are used by those records and that give you the flexibility to develop record parts of your own

#### functions.egl

Contains two sets of functions:

- MQSeries command functions, which access MQSeries directly
- Initialization functions, which let you place initial values in the options records that are used in your program

#### mqrcode.egl, mqrc.egl, mqvalue.egl

Contains a set of EGL dataTable parts that are used by the command and initialization functions

Your tasks are as follows:

1. Using the process for importing files into the workbench, bring those files into an EGL project. The files reside in the following directory:

```
installationDir\egl\eclipse\plugins\  
com.ibm.etools.egl.generators_version\MqReusableParts
```

*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The latest version of the plugin; for example, 6.0.0

2. To make the parts more easily available to your program, write one or more *EGL* import statements in the file that contains your program. If the files to be imported reside in a project other than the one in which you are developing code, make sure that your project references the other project.

For details, see *Import*.

3. In your program, declare global variables:
  - Declare MQRC, MQRCODE, and MQVALUE, each of which must use as a typedef the dataTable part that has the same name as the variable.
  - For each options record that you wish to pass to MQSeries, declare a basic record that uses an options record part as a typedef. For details on each part, see *Options records for MQ records*.

4. In your function, initialize the options records that you intend to pass to MQSeries. You can do this easily by invoking the imported EGL initialization function for a given options record. The name of each function is the name of the part that is used as a typedef for the record, followed by `_INIT`. An example is `MQGMO_INIT`.
5. Set values in the options records. In many cases you set a value by assigning an EGL symbol that represents a constant, each of which is based on a symbol described in the MQSeries documentation. You can specify multiple EGL symbols by summing individual ones, as in this example:

```
MQGMO.GETOPTIONS = MQGMO_LOCK
                  + MQGMO_ACCEPT_TRUNCATED_MSG
                  + MQGMO_BROWSE_FIRST
```

6. The first time you generate a particular program that uses the customization features of MQSeries support, you also generate the data tables used by that program. To generate all data tables that are used in your program, allow the build descriptor option **genTables** to default to YES. For additional details, see `DataTable` part.

## MQSeries-related EGL keywords

When you work with the MQSeries-related EGL keywords like *add* and *get next*, you define an MQ record for each message queue you wish to access. The record layout is the format of the message.

The next table lists the keywords.

Keyword	Purpose
add	<p>Places the content of an MQ record at the end of the specified queue.</p> <p>The EGL add statement invokes as many as three MQSeries commands:</p> <ul style="list-style-type: none"> <li>• MQCONN connects the generated code to a queue manager and is invoked when no connection is active.</li> <li>• MQOPEN establishes a connection to a queue and is invoked when a connection is active but the queue is not open.</li> <li>• MQPUT puts the record in the queue and is always invoked unless an error occurred in an earlier MQSeries call.</li> </ul> <p>After adding an MQ record, you must close a message queue before reading an MQ record from the same queue.</p>
close	<p>Relinquishes access to the message queue that is associated with an MQ record.</p> <p>The EGL close statement invokes the MQSeries MQCLOSE command, which also is invoked automatically when your program ends.</p> <p>You should close the message queue after an add or scan if another program requires access to the queue. The close is particularly appropriate if your program runs for a long time and no longer needs access.</p>

Keyword	Purpose
get next	<p>Reads the first message in a queue into a message queue record and (by default) removes the message from the queue.</p> <p>The EGL scan statement invokes as many as three MQSeries commands:</p> <ul style="list-style-type: none"> <li>• MQCONN connects the generated code to a queue manager and is invoked when no connection is active.</li> <li>• MQOPEN establishes a connection to a queue and is invoked when a connection is active but the queue is not open.</li> <li>• MQGET removes the record from the queue and is always invoked unless an error occurred in an earlier MQSeries call.</li> </ul> <p>After reading an MQ record, you must close the queue before adding an MQ record to the same queue.</p>

## Manager and queue specification

When you work with the MQSeries-related EGL keywords, you identify a queue in the following situations:

- At declaration time, you specify a logical queue name, and you do so by setting the **queueName** property of the MQ record part. That logical queue name acts as a default for the queue name accessed at run time; but in most cases the name is meaningful only as way of associating the MQ record with a physical queue. The logical queue name can be no more than 8 characters.
- At generation time, you control the generation process with a buildDescriptor part that in turn can reference a resource associations part. The resource associations part associates the queue name with the name of a physical queue.
- At run time, your code can change the value in the record-specific variable **record.resourceAssociation** to override any queue name you specified at declaration or generation time.

The name of the physical queue has the following format:

*queueManagerName:physicalQueueName*

*queueManagerName*

Name of the queue manager; if this name is omitted, the colon is omitted, too

*physicalQueueName*

Name of the physical queue, as known to the specified queue manager

The first time that you issue an add or scan statement against a message queue record, a connecting queue manager must be specified, whether by default or otherwise. In the simplest case, you do not specify a connecting queue manager at all, but rely on a default value in the MQSeries configuration.

The record-specific variable **record.resourceAssociation** always contains at least the name of the message queue for a given MQ record.

## Remote message queues

If you want to access a queue that is controlled by a remote queue manager, you must do the following:

- Issue the EGL close statement to relinquish access to the queue now in use
- Set the record-specific variable **record.resourceAssociation** to ensure later access of the remote queue



You set ***record.resourceAssociation*** in one of two ways, depending on how the queue-manager relationships are established in MQSeries:

- If the connecting queue manager has a local definition of the remote queue, set ***record.resourceAssociation*** as follows:
  - Accept the same value for the connecting queue manager (either by specifying the name of the connecting queue manager or by specifying no name; in the latter case, omit the colon).
  - Specify the name of the local definition of the remote queue.
 Your next use of the *add* or *scan* statement issues an MQOPEN to establish access to the remote queue.
- Alternatively, set ***record.resourceAssociation*** with the name of the remote queue manager, along with the name of the remote queue. The connecting queue manager does not change in this case. Your next use of the *add* or *get next* statement issues MQOPEN and uses the connection already in place.

#### Related concepts

“Direct MQSeries calls”

“MQSeries support” on page 336

#### Related reference

“MQ record properties” on page 772

“Options records for MQ records” on page 772

## Direct MQSeries calls

You can use a set of installed EGL functions that mediate between your code and MQSeries, as described in *MQSeries support*.

The next table lists the available functions and identifies the required arguments. MQBACK (MQSTATE), for example, indicates that when you invoke MQBACK you pass an argument that is based on the MQSTATE record part. The record parts are described later.

MQSeries-related EGL function invocation	Effect
MQBACK (MQSTATE)	Invokes the system function sysLib.rollback to rollback a logical unit of work. The rollback affects <i>all</i> recoverable resources being accessed by your program, including databases, message queues, and recoverable files.
MQBEGIN (MQSTATE, MQBO)	Begins a logical unit of work.
MQCHECK_COMPLETION (MQSTATE)	Sets the mqdescription field of the record that is based on MQSTATE. The setting is based on the last-returned reason code. The function MQCHECK_COMPLETION is called automatically from the EGL functions MQBEGIN, MQCLOSE, MQCONN, MQCONNx, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQPUT1, and MQSET.
MQCLOSE (MQSTATE)	Closes the message queue to which MQSTATE.hobj refers.



MQSeries-related EGL function invocation	Effect
MQCMIT (MQSTATE)	Invokes the system function sysLib.commit to commit a logical unit of work. The commit affects <i>all</i> recoverable resources being accessed by your program, including databases, message queues, and recoverable files.
MQCONN (MQSTATE, qManagerName)	Connects to a queue manager, which is identified by qManagerName, a string of up to 48 characters. MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls. <b>Note:</b> Your code can be connected to one queue manager at a time.
MQCONN(MQSTATE, qManagerName, MQCNO)	Connects to a queue manager with options that control the way that the call works. The queue manager is identified by qManagerName, a string of up to 48 characters. MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls.
MQDISC (MQSTATE)	Disconnects from a queue manager.
MQGET(MQSTATE, MQMD, MQGMO, BUFFER)	Reads and removes a message from the queue. The buffer cannot be more that 32767 bytes, but that restriction does not apply if you are using the EGL get next statement.
MQINQ(MQSTATE, MQATTRIBUTES)	Requests attributes of a queue.
MQNOOP()	Used only by EGL.
MQOPEN(MQSTATE, MQOD)	Opens a message queue. MQSeries sets the queue handle (MQSTATE.hobj) for use in subsequent calls.
MQPUT(MQSTATE, MQMD, MQPMO, BUFFER)	Adds a message to the queue. The buffer cannot be more that 32767 bytes, but that restriction does not apply if you are using the EGL <b>add</b> statement.
MQPUT1(MQSTATE, MQOD, MQMD, MQPMO, BUFFER)	Opens a queue, writes a single message, and closes the queue.
MQSET(MQSTATE, MQATTRIBUTES)	Sets attributes of a queue.

The next table lists the options records that are used as arguments when you invoke the MQSeries-related EGL functions. Also listed is the initialization function that should invoked for a given argument.

Your first step is to initialize the argument that is based on the MQSTATE record part. In the following example (as in the table that follows), the argument name is assumed to be the same as the name of the record part:

```
MQSTATE_INIT(MQSTATE);
```

Argument (the record part name)	Initialization function	Description
MQATTRIBUTES	none	Arrays of attributes and attribute selectors, plus other information used in the command MQINQ or MQSET
MQBO	MQBO_INIT (MQBO)	Begin options

Argument (the record part name)	Initialization function	Description
MQCNO	MQCNO_INIT (MQCNO)	Connect options
MQGMO	MQGMO_INIT (MQGMO)	Get-message options
MQIIH	MQIIH_INIT (MQIIH)	IMS information header; describes information that is required at the start of an MQSeries message sent to IMS (MQSeries documentation indicates that use of this header is not supported on Windows 2000/NT/XP)
MQINTATTRS	none	Arrays of integer attributes for use in the command MQINQ or MQSET
MQMD	MQMD_INIT (MQMD, MQSTATE)	Message descriptor (MQSeries version 2)
MQMDE	MQMDE_INIT (MQMDE, MQSTATE)	Message descriptor extension (use only the fields that are in MQSeries version 2)
MQOD	MQOD_INIT (MQOD)	Object descriptor
MQOO	MQOO_INIT (MQOO)	Open options
MQPMO	MQPMO_INIT (MQPMO)	Put-message options
MQSELECTORS	none	An array of attribute selectors, used only if you wish to access MQSeries without use of EGL functions
MQSTATE	MQSTATE_INIT (MQSTATE)	A collection of arguments that are each used in one or more calls to MQSeries; for example, when you connect with the EGL function MQCONN or MQCONNEX, MQSeries sets the connection handle (MQSTATE.hconn) for use in subsequent calls
MQXQH	MQXQH_INIT (MQXQH, MQSTATE)	Transmission queue header

**Note:** The record parts each contain only one structure item, and the structure item uses a structure part as a typeDef. This setup gives you maximum flexibility. You can create your own record parts that are each composed of a series of structure parts.

The name of each structure part is the name of the record part followed by `_S`; the record part MQGMO, for example, uses a structure part named MQGMO\_S.

#### Related concepts

“MQSeries-related EGL keywords” on page 339

"MQSeries support" on page 336  
"Record parts" on page 135  
"Typedef" on page 28

**Related reference**

"get next" on page 701  
"commit()" on page 1024  
"rollback()" on page 1036

---

## IMS runtime support

EGL allows you to develop programs that run on IMS, whether in MPP or BMP regions, whether as FastPath or non-FastPath programs, whether conversational or nonconversational:

- When you generate a textUI program for IMS/VS, EGL runtime handles the details of message-queue interaction that involves the I/O PCB. You also can use an alternate PCB to send output to a destination other than the originating terminal.

You can interact with the terminal by using the **converse** statement, which presents a text form and responds to the user's input by processing the statement that follows the **converse** statement. For an overview of the runtime behavior, see *Segmentation in text applications* and, especially, *Behavior of a segmented program on CICS and IMS*.

Although use of a **converse** statement is relatively simple, you get better performance by using a **show** statement that returns to the beginning of the same program. Use of a **show** statement requires a more complicated design because the re-invoked program starts at the first line, and that initial code must analyze whether the program is being invoked at the beginning or in the middle of a user-code interaction.

- When you generate a basic main program for IMS/VS, no text forms are involved, but you can accept input from an input IMS message queue. You handle the details of message-queue interaction by coding a loop that reads messages into a serial record which you associate with the queue. You also can use an alternate PCB to send output.

If a non-conversational EGL-generated program transfers control to an EGL-generated basic program by using a transfer statement of the form *transfer to a transaction*, and if the statement also transfers a record, the transferred record is not used to initialize the receiving program's input record, as is the case in other environments. Instead, the receiving program must read the transferred record from the message queue.

- When you generate a basic main program for IMS BMP, you can accept input from an input IMS message queue. You handle the details by coding a loop that reads from a serial record which you associate with that queue. You also can use an alternate PCB to send output, and you can use a serial record and a GSAM PCB record to interact with a GSAM file.
- When you generate a called program for IMS/VS or IMS BMP, you do not read from the message queue, but can use an alternate PCB to send output; and in the case of IMS BMP, you can use a serial record and a GSAM PCB record to interact with a GSAM file.

For details on how to interact with IMS control blocks, see *EGL support for runtime PCBs and PSBs*.

### Related concepts

"Behavior of a segmented program on CICS or IMS" on page 190

"DL/I database support" on page 310

"EGL support for runtime PSBs and PCBs" on page 346

"Segmentation in text applications" on page 189

### Related tasks

"Interacting with terminals in IMS" on page 349

"Using serial files in IMS" on page 353

### Related reference

"add" on page 661

"converse" on page 672

"show" on page 751

"transfer" on page 752

---

## EGL support for runtime PSBs and PCBs

To interact with IMS message queues, DL/I databases, and GSAM files, you customize EGL program elements as needed to generate a COBOL program that can access your organization's program specification blocks (PSBs) and program communication blocks (PCBs). Those blocks are hereafter called the runtime PSBs and runtime PCBs.

First, define the DLI`Segment` record parts that you will reference in database PCB records (if any). Your primary, IMS-specific tasks are then as follows:

1. Define a PSB record part. That part includes the set of PCB records that will be used when accessing IMS resources.
2. In the program, make the PSB and PCB information available:
  - Declare a record that is based on the PSB record part
  - In the set-value block for the program, set the property `@DLI`, property field `psb`, to the name of the PSB record

Each PCB record is based on one of the following, predefined PCB record parts:

### IO\_PCBRecord

Used to interact with an I/O PCB, which allows for input from a program or terminal and (if the input came from a terminal) allows for output to the same terminal. The I/O PCB also provides access to other IMS capabilities; for example, checkpoint and restart of a batch program.

### ALT\_PCBRecord

Used to reference a teleprocessing PCB other than the I/O PCB. This type of record allows your code to write output to a message queue that is associated either with another transaction or with a device other than the terminal associated with the I/O PCB. The runtime PCB may be either of the following kinds:

- An alternate PCB, in which case the output occurs only if a commit occurs; or
- An express alternate PCB, in which case the output occurs regardless of whether a commit or rollback occurs.

### DB\_PCBRecord

Used to reference a database PCB, which represents a DL/I database accessible from your program. The runtime database PCB specifies the data that can be accessed and the type of access that is valid.

### GSAM\_PCBRecord

Used to reference a GSAM PCB, which is used when a z/OS batch or IMS BMP program accesses a serial file that acts as a root-only DL/I database.

The next list provides details on the runtime PSB in each of the target systems.

## CICS

The value of the PSB record property **defaultPSBName** is (by default) the name of the runtime PSB. EGL places that name in the **psbName** field of the system variable **DLILib.psbData**, but you can assign a different value to that library field. When your program attempts an I/O operation against a DL/I database, the value in **psbName** determines what PSB is used.

The system variable **DLILib.psbData** has a second field, **psbRef**. The initial value of the field is zero, which indicates that no PSB is scheduled. When CICS schedules the PSB, EGL runtime assigns to that field the address used to access the PSB. You can use the variable to "pass the PSB" (really, to pass a name and the related address) during a transfer or call.

**Note:** You must avoid writing logic that assigns any value to **DLILib.psbData.psbRef**.

An I/O PCB is valid in the runtime PSB (to allow for checkpoints in DL/I access, for example); alternate PCBs are tolerated; and DB PCBs are valid.

## IMS BMP

The PSB parm in the runtime JCL identifies the runtime PSB used throughout the job step. Although you can customize the JCL at deployment time, EGL generates the default PSB parm value in the runtime JCL by assigning the value of the PSB record property **defaultPSBName**.

For an IMS BMP, the first runtime PCBs are as follows:

1. The I/O PCB (created if your IMS system programmer sets CMPAT to YES when developing the PSBGEN job)
2. An alternate PCB
3. An express alternate PCB

DB and GSAM PCBs are also valid.

## IMS/VS

The rules of IMS system definition ensure that the name of the main program is the name of the runtime PSB, which is available throughout the transaction.

The first runtime PCBs are as follows--

1. The I/O PCB (created if your IMS system programmer sets CMPAT to YES when developing the PSBGEN job)
2. An alternate PCB
3. An express alternate PCB

DB PCBs are also valid.

If the value of build descriptor option **workDBType** is DLI (as is the default), set your last DB PCB for the EGL work database, which is identified by the name ELAWORK.

## z/OS batch

The PSB parm in the runtime JCL identifies the runtime PSB used throughout the job step. Although you can customize the JCL at deployment time, EGL generates the default PSB parm value by assigning the value of the PSB record property **defaultPSBName**.

An I/O PCB is valid in the runtime PSB (to allow for checkpoints in DL/I access, for example); alternate PCBs are tolerated; and DB PCBs are valid.

EGL adjusts for the initial two or three I/O and teleprocessing PCBs if they are declared in the PSB record but are not present in the runtime PSB. This adjustment allows you to generate the same program across different environments. In relation to CICS, for example, EGL runtime ignores the initial I/O and alternate PCB records if they are present in the code but do not match the runtime PSB used in CICS.

## Requirements for the PSB record part

You specify a call interface (AIBTDLI or CBLTDLI) by setting the program property **@DLI**, property field **callInterface**, and your choice affects the requirements for the PSB record part that you define:

- If (as is recommended), you set the **callInterface** field to AIBTDLI, access to a given runtime PCB is by PCB name, and the structure of the PSB record in your program does not need to reflect the structure of a runtime PSB. The following requirements are in effect for main programs, however:
  - For IMS/VS or IMS BMP, the first PCB in the runtime PSB must be an I/O PCB and must be called IOPCB; and the name of the first PCB record in your PSB record part must be IOPCB.
  - For any EGL-generated program that runs in IMS and that uses an alternate PCB, EGL must be able to use the name ELAALT to access that kind of runtime PCB. You must take one of the following actions:
    - Name the PCB record with the required name; or
    - Do as follows:
      - Use the required name (in this case, ELAALT) as the name of the PCB record; and
      - Use the runtime PCB name as the value of the record property **@PCB**, property field **PCBName**.

Alternatively, you can declare a record that redefines the alternate PCB record, and name that overlay record ELAALT. For details on redefining records, see *Declaring a record that redefines another*.

- For any EGL-generated program that runs in IMS and that uses an express alternate PCB, EGL must be able to use the name ELAEXP to access that kind of PCB. Your choices are equivalent to those listed for the alternate PCB.
- Finally, if you set the build descriptor option **workDBType** to DLI (the default value) when generating a program for IMS/VS, EGL runtime uses a DL/I work database and must be able to access a database PCB named ELAWORK. Your choices are equivalent to those listed for the alternate PCB.

In a called program, you need to declare only the PCB records that are used there.

- If you set the **callInterface** field to CBLTDLI, access to a given runtime PCB is by address rather than by name.

With the exception of I/O and alternate PCB records that EGL ignores so you can generate the same program across different environments, the structure of the PSB record in a main program must reflect at least the initial PCBs in the runtime PSB--

- The PSB record cannot have more PCB records than the number of PCBs in the runtime PSB but can have fewer
- The position of each PCB record must match the position of the related runtime PCB

The situation in called programs is as follows--

- If you pass a PSB record to the called program, you are passing an address used to access the runtime PSB. You must set up at least the initial part of the PSB record part as you did in the main program, including the PCB records for the I/O, alternate, and alternate express PCBs (if used in a particular environment), as well as other PCB records needed in the called program. You also must set the program property **@DLI**, property field **psbParm**.
- If you pass PCB records to the called program (as is preferred), you are passing addresses used to access each runtime PCB. You still must set up at least the I/O, alternate, and alternate express PCBs (if used in a particular environment); but aside from those, you need to declare only the PCB records that are needed in the called program. You also must set the program property **@DLI**, property field **pcbParms**.

If you specify properties **pcbParms** and **psbParm**, the PCB-specific addresses in the former override the equivalent addresses in the latter; the passed PSB record is ignored.

#### **Related concepts**

“DL/I database support” on page 310

“IMS runtime support” on page 345

#### **Related reference**

“add” on page 661

“converse” on page 672

“show” on page 751

“transfer” on page 752

#### **Related tasks**

“Declaring a record that redefines another” on page 54

“Interacting with terminals in IMS”

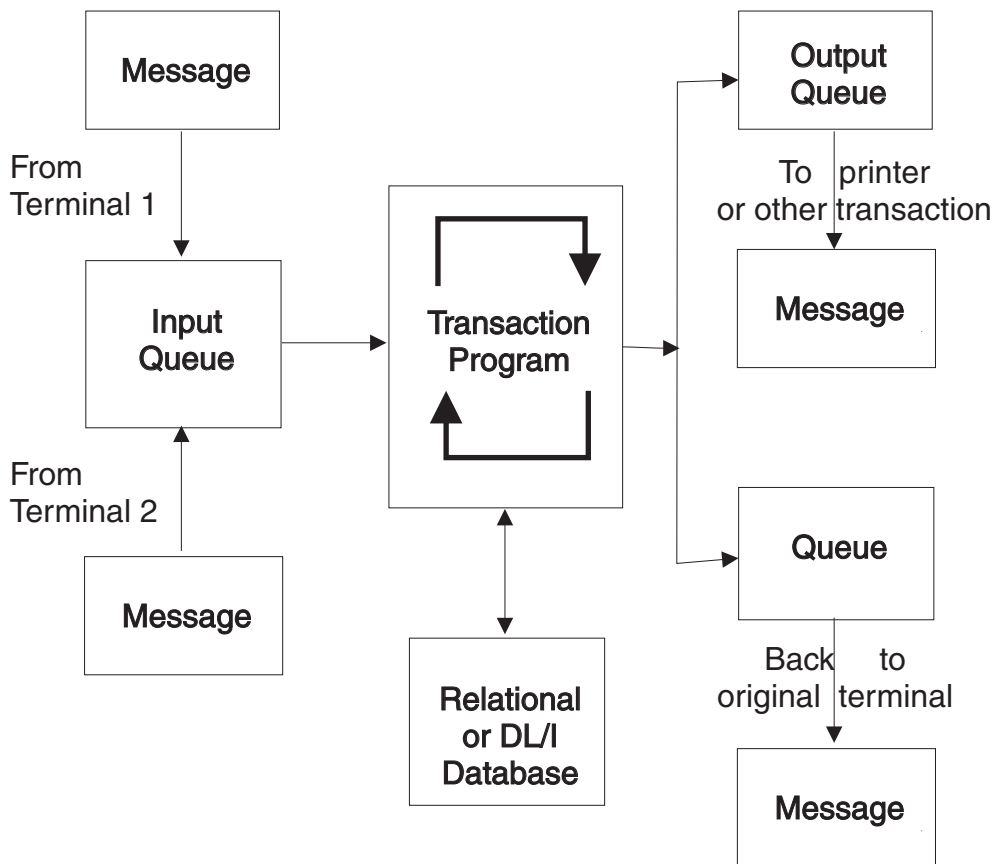
“Using serial files in IMS” on page 353

---

## **Interacting with terminals in IMS**

Typical IMS programs use a message-driven structure like that in the figure below:





In this example, the IMS controller starts the transaction program when the message queue associated with the program contains a message. Another program might have put the message on the queue, or the controller might have read input from the terminal. The program takes the message off the queue, does any required database I/O, and adds messages to output queues to continue processing. The output queue can represent the input terminal, another terminal or printer, or a queue associated with another transaction. The program then loops back to the beginning and processes the next message on its input queue.

Typical PL/I or COBOL programs must continue the cycle until the message queue is empty because multiple terminals run the same transaction concurrently. However, EGL programs automatically loop to read the next message in the queue. You do not need to define message queue control functions directly. You can define programs for IMS just as you define programs for CICS, that use a synchronous logic structure instead of a message-driven structure. With the synchronous model, you only need to consider the processing that must occur for a single user at a single terminal. This simplifies both the design and the definition of the program.

IMS requires you to commit all database changes and to release all database locks and positions when waiting for user input. In EGL, this means creating a segmented program (or a single-segment program). When you define the program, remember that EGL performs a commit with each **converse** I/O statement. You must understand how segmentation works to develop programs for IMS; see *Segmentation in text applications*.

#### Related concepts

"DL/I database support" on page 310

“Segmentation in text applications” on page 189

#### Related reference

“add” on page 661

“converse” on page 672

“get” on page 687

#### Related tasks

---

## Defining forms for IMS programs

EGL generates your FormGroup parts into IMS Message Format Services (MFS) maps. Each form that you define for use in the IMS environment must provide space for the following:

- An 8-byte constant field that is used to store the IMS transaction name
- A 2-byte area that EGL uses to record the types of information stored in the work database

You can define the 8-byte constant field with the protect and dark attributes. The attribute byte on the form becomes the attribute byte in the EGL-generated MFS map. The 8-byte constant contains the name of the IMS transaction that is started when the form is processed. Specifying the constant on the form enables the user to specify the IMS `/FORMAT` command to display a formatted screen to start a transaction. Do not use the `/FORMAT` command if variable fields on the form have initial default values. If the `/FORMAT` command is used, the default values do not appear.

If you do not define an 8-byte, protected, dark constant on the form, EGL searches for any string of 9 blanks on the form and sets this area aside as a protected, dark variable field (1 byte attribute, 8 bytes of data) in the generated MFS map. The generated program uses this field to store the name for the next IMS transaction to be run after a **converse** statement, or after a **transfer** statement of the form *transfer to a transaction* that includes a map. The user cannot use the `/FORMAT` command to start a transaction for these maps because IMS does not have a default transaction name.

You do not need to explicitly define the 2-byte area on a form. EGL selects two adjacent blank bytes on the map and treats it as a protected, dark variable field (1 byte attribute, 1 byte of data).

## Estimating the size of MFS blocks for a formGroup

When EGL generates a formGroup, it generates the MFS control blocks for that formGroup. There are three types of MFS control blocks:

#### Device input format (DIF) and device output format (DOF)

These control blocks describe the arrangement of data fields and literals in the device presentation space (for example, the screen for 3270 devices).

For 3270-type devices, a single set of statements describes both the DIF and the DOF. For printers, only a DOF is needed. Each device field is given a name that statements can refer to in the message input and output descriptors.

For EGL FormGroup parts, the DOF is always larger than the DIF because the DOF includes form constants.

#### Message output descriptor (MOD)

This control block describes the various fields of information in the output

message inserted by the program. It also identifies corresponding device fields where the data for each message field is moved.

### **Message input descriptor (MID)**

This control block describes the various fields of information in the input message retrieved by the program. The MID identifies the corresponding device field from which the data for each message field came.

MFS control blocks cannot exceed 32748 bytes. If you are using a large FormGroup part, the following formulas offer a guideline for estimating an upper limit for the size of the control blocks that will be generated. Using these formulas during your design helps you determine whether your FormGroup parts should be split into smaller ones. If a generated control block is too large, MFS generation issues a 3022 abnormal termination.

### **Calculating the DOF size for display devices**

The following formula helps you estimate the size of the DOF:

DOF Size =  
150  
+ 388 \* Number of printer forms in the FormGroup  
+ 208 \* Number of display forms in the FormGroup  
+ 63 \* Number of variable field occurrences on display forms in the FormGroup  
+ 62 \* Number of constant fields on display forms in the FormGroup  
+ 1.12 \* Total length of all constant fields on display forms in the FormGroup

### **Calculating the DOF size for printer devices**

The following formula helps you estimate the size of the DOF:

DOF Size =  
206  
+ 68 \* Number of printer forms in the FormGroup  
+ 374 \* Number of display forms in the FormGroup  
+ 63 \* Number of variable field occurrences on printer forms in the FormGroup  
+ 62 \* Number of constant fields on printer forms in the FormGroup  
+ 1.12 \* Total length of all constant fields on printer forms in the FormGroup

### **Calculating the MOD size for all forms**

The following formula helps you estimate the size of the MOD:

MOD Size =  
36  
+ 724 \* Number of display forms in the FormGroup  
+ 202 \* Number of printer forms in the FormGroup  
+ 52 \* Number of variable field occurrences in the FormGroup

### **Calculating the MID size for terminal maps**

The following formula helps you estimate the size of the MID for terminal maps:

MID Size =  
36  
+ 858 \* Number of display forms in the FormGroup  
+ 52 \* Number of variable field occurrences for display forms in the FormGroup

### **Related concepts**

“IMS runtime support” on page 345

---

## Using serial files in IMS

Serial files must be implemented as IMS message queues in IMS/VS. They can be implemented as message queues, OS/VS files, VSAM files, or GSAM files for IMS BMP. Serial files can be implemented as OS/VS files, VSAM files, or GSAM files for z/OS batch. The following discussion deals with using GSAM files or message queues for serial files.

EGL programs that run in the IMS BMP or z/OS batch environments can implement serial files as GSAM files. You can use the **add**, **get next**, and **close** I/O statements for serial files that you implement as GSAM files. Here is a list of differences between GSAM and normal serial file processing:

- A GSAM file requires a DBD.
- A GSAM file requires a PCB in the IMS PSB. You must define this PCB in the IMS runtime PSB and in the EGL PSB definition. In your program, you must declare a record variable that is based on the PSB record part.
- A GSAM file is read or written through DL/I calls. The generated COBOL program handles this automatically, based on the I/O statements that you request.
- A GSAM file is checkpointed and restarted in the same way as a DL/I database. However, to recover the GSAM file requires the use of symbolic checkpoint and restart instead of basic checkpoint.

EGL does not support the record search argument for GSAM or undefined length records. You identify a serial file or printer file as a GSAM file by using the resource association part during generation to specify a file type of GSAM and a PCB name.

When you associate a serial file with a GSAM file, you must include the following information:

### Resource name

Indicates the 1- to 44-character data set name that is used in the sample runtime JCL. The file name from the record definition is used as the DD name in the sample runtime JCL.

### File type

Specifies GSAM as the file type to associate the serial file or printer output with a GSAM file.

### PCB name

Specifies a PCB name for the serial file that is associated with the GSAM file. If you do not specify one, the default is the first GSAM PCB in the EGL PSB.

## Using serial files as message queues

Online programs that run in IMS/VS implement serial files as IMS message queues. Programs that run as IMS BMP programs can also implement serial files as message queues. You can use the **add** and **get next** I/O statements as well as **close** for output files. If you select IMS/VS or IMS BMP as the target runtime environment, you can define serial or print files as being associated with a message queue. You must associate all serial files and print files with message queues for IMS/VS. Only a single input file can be associated with the message queue.

You can associate a serial file or printer file with a message queue by using a resource association part during generation and specifying the file type and a PCB name. When you associate a serial file with a message queue, you must define the following resource information:

#### Resource name

You must indicate the 1- to 8-character destination ID for printer or serial file data. The name must match the ID of an IMS logical terminal or a transaction code that is defined in the IMS system definition.

The file name is the default resource name for the message queue. You can override this default in the resource association part.

If the PCB that you select is a modifiable alternate or express alternate PCB, you can override the default message queue name at run time by setting a value for `sysVar.resourceAssociation` for a file or `converseVar.printerAssociation` for a printer in the program. `sysVar.resourceAssociation` is treated as a local variable. Setting `sysVar.resourceAssociation` for a record in one program does not affect `sysVar.resourceAssociation` in another program. An **add** statement writes to the message queue identified by the setting of `sysVar.resourceAssociation` for that program.

#### Message queue type

You can specify single-segment message queues or multiple-segment message queues.

##### Single-segment message queues (SMMSGQs)

For a single-segment message queue, each record that you add or that you read (with a **get next**) from the serial file is a complete message. The generated COBOL program issues an IMS PURG call between records that are added to a single-segment message queue. The generated COBOL program issues an IMS get unique for each **get next** statement.

##### Multiple-segment message queues (MMSGQs)

For multiple-segment message queues, a series of adds to the serial file is treated as though each **add** statement were for a segment of a single message. The message is not ended until you issue a **close** statement or reach a commit point. The generated COBOL program issues an IMS PURG call for the **close** statement. You can then begin adding segments of another message and close it. Multiple-segment message queues are not valid for printer files.

If you issue a **get next** statement for a MMSGQ serial file, the generated program issues an IMS get unique call to get the first segment of the message. Additional **get next** statements result in GN calls to get the remaining segments of the message. At the end of all the segments in a message, the generated COBOL program sets the `noRecordFound` record state. If you continue scanning, the generated program starts another series of get unique (GU) calls, followed by get next (GN) calls. When no more messages are found, the generated program returns an `endOfFile` state.

#### PCB name

You must also specify a PCB name for the serial file that is associated with a message queue. You must specify the name assigned to the I/O PCB as the PCB name for a serial input file. The I/O PCB is the only message queue used for input. If you use a serial input file, you must use a main batch or called batch program. The generated program handles all I/O PCB logic for main transaction programs.

You can specify the PCB name for a serial output file. The PCB name must be the name assigned to an alternate PCB record. The default PCB name is the name of the first alternate PCB in the PSB. You can only send output to the I/O PCB by using one of the following system functions:

- VGLib.VGTDLI()
- DLILib.AIBTDLI()
- DLILib.EGLTDLI()

## Defining records to use with message queues

When you define a serial record to associate with a message queue, you should define only the program data. The generated COBOL program adds the IMS message header (length, ZZ, and transaction code) for an **add** statement and removes it for a **get next** statement.

## Checking the results of serial file I/O statements

When a serial file is associated with a message queue or GSAM database, the generated program issues a DL/I call to implement the I/O operation. When the DL/I call completes, Enterprise Developer Server for z/OS performs the following functions:

- For **get next** statements, the record state is set based on the DL/I status code. The sysVar.sessionID or sysVar.userID field is updated from the user ID field of the I/O PCB when the generated program issues a GU call for the I/O PCB. This happens at the first **get next** statement for a serial file defined as a multiple-segment message queue (MMSGQ), and at each **get next** statement for a single-segment message queue (SMSGQ). The EGL sysVar.transactionID field is updated from the transaction name in the IMS message header after each **get next** statement that results in a GU call for the I/O PCB.
- For an **add** or **close** statement, the record state is updated based on the DL/I status code.

After a DL/I call that involves either the message queue or GSAM, the DLIVar fields are not updated. These fields are updated only for functions that access DL/I segment records. This allows a program written for a CICS transient data queue or an OS/VS serial file to run consistently when the file is changed to a message queue or GSAM database in an IMS environment. You should check the I/O error values to determine if End Of File was reached or an error occurred on the serial file. If you need more detailed information from the PCB, use the field names in the IO\_PCBRecord or the ALT\_PCBRecord. Consider a situation in which your PSB variable (named myPSB) declares an ALT\_PCBRecord named myAltPCB, and you used myAltPCB as the PCB name in your resource allocation. To reference the DL/I status code after an **add** statement, use myPSB.myAltPCB.statusCode.

## Using printer files as message queues

For IMS/VS you must associate printer files with message queues. For IMS BMP, you can associate printer files with message queues. You associate printer files with message queues the same way you associate serial files with message queues, with the exception that SMSGQ is the only valid file type for a resource association whose file name is *printer*. In your IMS system definition, you must define the message queue name that you use in the runtime environment as a logical terminal. converseVar.printerAssociation can be used to change the printer destination at run time. For example, you could define a table of user IDs and the printer ID that each user normally uses. By setting converseVar.printerAssociation, you can route the printer output to a printer near the program user.

#### Related concepts

"DL/I database support" on page 310

"Record types and properties" on page 138

"Segmentation in text applications" on page 189

#### Related reference

"add" on page 661

"converse" on page 672

converseVar.printerAssociation

"close" on page 669

"get" on page 687

sysVar.resourceAssociations

#### Related tasks

"Interacting with terminals in IMS" on page 349

---

## Calling an IMS program from EGL-generated Java code

You can call an IMS program remotely, from EGL-generated Java code. The called program can be generated from EGL or VisualAge Generator or can be written in another language.

The program accessed on IMS is not the called program itself, but is a catcher program provided by Enterprise Developer Server for z/OS. As shown later, the system programmer must re-link that catcher program. The effect of that task is to assign an alias for each runtime PSB associated with any transactions that are invoked remotely by EGL-generated Java code.

The runtime process is as follows:

1. EGL runtime takes the name of the IMS transaction code from the linkage options part used at generation time of the calling program.
2. EGL uses the connectors of IMS Connect to submit this transaction code, as well as the called program name and parameters, to the IMS message queue.
3. The catcher program reads the called program name and parameters from the message queue and uses a z/OS call to invoke the requested program
4. On regaining control, the catcher program submits the returned data to the IMS queue
5. IMS Connect reads the data from the queue and returns the data to the calling program

Here is an example:

1. On IMS, the systems programmer carries out the following tasks:
  - a. Creates a system definition that associates a transaction (for example, TRAN1) with a PSB (for example, PSB1).
  - b. Links the catcher program ELAISVN to assign it the alias PSB1. The linkage can include up to 64 aliases of this kind, and the name of the module name can be any you choose. If you wish to add an alias after 64, create a second load module.
2. You place a statement in your Java program to call PGMX and to supply parameters for that program.
3. In the build descriptor used to generate the program, you set the **linkage** build descriptor option to a linkage options part called **pgmLinkage**



4. In that linkage options part, you set the **callLink** element, **serverID** property to the appropriate transaction code (in this case, to TRAN1).
5. At run time, IMS Connect sends the transaction code (TRAN1), program name (PGMX), and parameters to the IMS message queue.
6. Because TRAN1 has been invoked, IMS schedules PSB1, which starts the catcher program.
7. The catcher program reads the message queue for the program name (PGMX) and parameters, then calls PGMX.
8. When PGMX finishes, control returns to the catcher program, which places the returned data on the IMS message queue.
9. IMS Connect returns the data to your Java code.

The transaction TRAN1 must be defined to IMS as a message processing program. Use the following IMS system definition as a model:

```
APPLCTN PGMTYPE=TP,PSB=PSB1
TRANSACT CODE=TRAN1,MODE=SNGL,EDIT=UL
```

Data will be folded to uppercase characters if the statement EDIT=ULC is omitted from the transaction definition.

Here is an example of the JCL the system programmer might use to re-link the catcher program ELAISVN, in this case to assign the aliases PSB1 and PSB2:

```
//L EXEC ELARLINK
//L.SYSLMOD DD DISP=SHR,DSN=load-library-name
//L.SYSIN DD *
INCLUDE SELALMD(ELAISVN)
ENTRY ELAISVN
ALIAS PSB1
ALIAS PSB2
NAME load-module-name(R)
/*
```

*load-library-name*

Name of the load library

*load-module-name*

Name of the load module; usually ELAISVN

IMS requires that the name of a runtime PSB be identical to the name (or, in this case, the alias) of the first program in a transaction. If you wish your called program to be called, not only from remote code, but by a program on IMS, you must do as follows:

1. Create a second PSB that is named for the first program in the additional run unit
2. Structure that PSB like the PSB scheduled for the remote invocation

#### **Related concepts**

“EGL support for runtime PSBs and PCBs” on page 346

“IMS runtime support” on page 345

“Linkage options part” on page 399

#### **Related reference**

“callLink element” on page 499



---

## Example IMS program code

These excerpts from EGL programs show interaction with IMS terminals, message queues, and serial files.

### Example of message queue I/O

Here are some code excerpts from an EGL program that accesses message queues. The program carries out the following tasks:

1. Requests input from a terminal using a form
2. Reads the response
3. Updates a serial record with information based on the user input
4. Outputs the serial record to another transaction for later batch processing

The program assumes the following associations in your IMS environment:

- IMS transaction code MYTRXCD1 is associated with a PSB named MYTRXCD1
- IMS transaction code NEXTTRX is associated with a PSB named MYTRXCD2

```
//define PSB
Record addToQueue type PSBRecord { defaultPSBName="MYTRXCD1" }
// three PCBs required for CBLTDLI on IMS
iopcb IO_PCBRecord { @PCB { pcbType = TP } };
elaalt ALT_PCBRecord { @PCB { pcbType = TP } };
elaexp ALT_PCBRecord { @PCB { pcbType = TP } };
// other database PCBs
...
end

Record myTransactionPart type serialRecord
{ fileName="MYMSGQUE" }
...
end

program addtrans type textUIProgram
{ alias = "MYTRXCD1",           // IMS requires pgm to match PSB name
  segmented = yes,
  @DLI { psb = "mypsbs" }}

use MYFORMS; // MYFORMS is a formGroup containing FORM1

// declare variables
myTransaction myTransactionPart; // serial record for message queue
mypsbs addToQueue;              // psb

function main()
  converse FORM1;
  // do whatever processing is necessary
  move FORM1 to myTransaction byName;
  add myTransaction;
end
end
```

When you generate, you must specify a resource association part that associates the serial file with a message queue and provides the name of the transaction to which it is to be sent and the name of the PCB to use, as in the following example:

```
<ResourceAssociations name="RESOURCEASSOC601_REORDER">
  <association fileName="MYMSGQUE">
    <imsvs>
```

```

        <smgq systemName="NEXTTRX" pcbName="elaalt"/>
    </imsvs>
</association>
</ResourceAssociations>

```

Because addtrans is a textUIProgram, EGL will make it behave properly in an IMS environment and read the message queue until the queue is empty. This means EGL will build a loop into the program such that once the program has finished placing the transaction into the serial file for later batch processing, it will return to the **converse** statement to request another transaction from the terminal.

## Example of IMS batch processing

You can also create a program to process the messages that program addtrans writes to the message queue. The program must be a basic program that gets records from a serial file and associates the serial file with the I/O PCB. The program can use the same serial record that addtrans used. Key changes show in bold in the example below:

```

//define PSB
Record getFromQueue type PSBRecord { defaultPSBName="MYTRXCD2" }
    // three PCBs required for CBLTDLI on IMS
    iopcb IO_PCBRecord { @PCB { pcbType = TP } };
    elaalt ALT_PCBRecord { @PCB { pcbType = TP } };
    elaexp ALT_PCBRecord { @PCB { pcbType = TP } };
    // other database PCBs
end

program gettrans type basicProgram
    { alias = "MYTRXCD2"
      @DLI { psb = "myspb" }}

// declare variables
myTransaction myTransactionPart; // serial record for message queue
myspb getFromQueue;           // psb

function main()
    while (myTransaction not endOfFile)
    get next myTransaction;
        // do whatever processing is necessary
    end
end
end

```

When you generate the program for either the IMS/VS or the IMSBMP environments, you must also specify a resource association part that associates the serial file with a message queue and provides the name of the transaction to which it is to be sent and the name of the PCB to use. In this case, the I/O PCB is used for input, as shown in the following example:

```

<ResourceAssociations name="RESOURCEASSOC601_REORDER">
    <association fileName="MYMSGQUE">
        <imsvs>
            <smgq systemName="NEXTTRX" pcbName="iopcb"/>
        </imsvs>
    </association>
</ResourceAssociations>

```

## Multiple users and message queues

The situation becomes more complex on IMS when you have multiple users in contention for the resources. Assume USER1 and USER2 both enter MYTRXCD1 on their terminals at the same time. Assume that USER1's transaction code ends up first on the message queue associated with MYTRXCD1.

1. IMS schedules the PSB associated with the transaction code MYTRXCD1. That PSB happens to be named MYTRXCD1 as well, though it does not have to be. The program associated with the PSB, however, must have the same name as the PSB in IMS. Therefore IMS loads the program MYTRXCD1 (known as addtrans to EGL).
2. The EGL-generated control logic in program MYTRXCD1 determines that this is the first time the program has been invoked for USER1, so processing begins at the top.
3. Eventually the program reaches the **converse** statement and performs the following actions:
  - Saves the data for all records and forms the program is using
  - Saves information about where in the program the **converse** statement occurred
  - Performs the ISRT command with the specified form
4. Following the logic that EGL added, the program loops back to the beginning and finds USER2 waiting on the message queue. The program follows the same steps for USER2 that it did for USER1, reaching the **converse** statement, sending the form, then looping back to check the message queue again.
5. Here the program is likely to find USER1's response to the **converse** statement. The EGL-generated control logic determines that this response is a continuation of USER1's processing, and does the following:
  - Restores USER1's data (including the location of the **converse** statement)
  - Refreshes a number of system variables
  - Runs any validation checks that FORM1 requested
  - Assuming no input errors, resumes processing at the statement after the **converse** statement
6. Eventually, the program reaches another **converse** statement, saves all data, and sends a response to USER1. The program then loops back to check the message queue again.
7. Assume that USER2 has gone to lunch and there is nothing left on the message queue associated with the transaction code MYTRXCD1. IMS shuts down the program MYTRXCD1.
8. Later, if USER1 responds to the most recent console message, IMS will once again have a message on the queue associated with transaction code MYTRXCD1 and will start the program MYTRXCD1 again. The EGL-generated control logic determines that this response is a continuation of USER1's processing, restores all data, and continues processing.

#### **Related concepts**

"DL/I database support" on page 310

#### **Related reference**

"add" on page 661

"audit()" on page 1018

"close" on page 669

"converse" on page 672

"get next" on page 701

#### **Related tasks**

"Interacting with terminals in IMS" on page 349

"Using serial files in IMS" on page 353

---

## IMS and DL/I error codes

`sysVar.errorCode` is set to any DL/I status code for GSAM and Message Queue I/O, just as for any other file I/O. EGL I/O error codes map to DL/I error codes as shown in the following table:

EGL I/O error code	DB status code	Severity
<code>endOfFile</code>	GB	Soft
<code>noRecordFound</code>	GE	Soft
<code>duplicate</code>	II (duplicates allowed)	Soft
<code>unique</code>	II (duplicates not allowed)	Hard
<code>ioError</code>	any non-blank status code	Hard or soft
<code>hardIOError</code>	non-blank other than GA, GB, GD, GE, GK, II	Hard
	<b>IMS Msg Queue status code</b>	
<code>endOfFile</code>	QC	Soft
<code>noRecordFound</code>	QD	Soft
<code>hardIOError</code>	non-blank other than QC, QD, CE, CF, CG, CI, CJ, CK, CL	Hard
	<b>GSAM status code</b>	
<code>endOfFile</code>	GB	Soft
<code>hardIOError</code>	non-blank other than GB	Hard

An exception of type `DLIException` is thrown for any hard error related to any type of DL/I access. For details, see *EGL system exceptions*.

### Related concepts

“DL/I database support” on page 310

### Related tasks

“EGL system exceptions” on page 587



---

## Maintaining EGL code

---

### Line commenting EGL source code

To comment one line of code, do as follows:

1. Click on the line, then right-click. A context menu is displayed.
2. Select **Comment**. Comment indicators (//) are placed at the beginning of the line.

To comment multiple consecutive lines of code, do as follows:

1. Click on the starting line. Holding down the left mouse button, drag the cursor to the ending line. Release the mouse button, and the range of lines is highlighted.
2. Right-click, then select **Comment** from the context menu. Comment indicators (//) are placed at the beginning of each of the lines in the selected range.

Use the same procedures to uncomment lines, but select **Uncomment** from the context menu.

#### Related tasks

"Creating an EGL source file" on page 130

"Opening a part in an .egl file" on page 367

#### Related reference

"EGL editor" on page 577

---

## Searching for parts

If you have a file open in the EGL editor, you can search for parts after setting the search criteria:

1. Open an EGL file. You cannot use the search facility unless the EGL editor is active; however, your search is not limited to the file that is open in the editor.
2. On the Workbench menu, click **Search > EGL**. The Search dialog is displayed.
3. If the EGL Search tab is not already displayed, click **EGL Search**. Notice that the conditions specified throughout the Search tab can affect the results.
4. Type the name of a part you want to locate; or to display a list of parts with names that match a specific pattern of characters, embed wildcard symbols in the name:
  - A question mark (?) represents any one character
  - An asterisk (\*) represents a series of any characters

For example, type *myForm?Group* to locate parts named myForm1Group and myForm2Group, but not myForm10Group. Type *myForm\*Group* to locate parts named myForm1Group, myForm2Group, and myForm10Group.

5. To make the search case-sensitive (so that myFormGroup is different from MYFORMGROUP), click the check box.
6. In the Search For box, select a type of part, or select **Any element** to expand your search to all part types.
7. In the Limit To box, select the option to limit your search to part declarations, part references, or both.

8. In the Scope box, select **Workspace** to search your workspace, **Enclosing Projects** to search the project that is currently highlighted in Project Explorer, or **Working Set** to search a defined set of projects. If you choose the Working Set scope, click the **Choose** button to select an existing working set or to define a new working set.
9. Click the **Search** button. The results of the search are displayed in the Search view.
10. If you double-click a file in the Search view, the file opens in the EGL editor, and the matching part is highlighted. If there is more than one match in the file, the first match is highlighted.  
Arrows in the left margin of the editor indicate the locations of each matching part.

#### Related concepts

"Parts" on page 19

#### Related tasks

"Opening a part in an .egl file" on page 367

#### Related reference

"EGL editor" on page 577

---

## Viewing part references

You can display a hierarchical view of the EGL parts that are referenced in a program, library, PageHandler, service, or report handler part; and you can access those parts:

1. Open the Parts Reference view in one of the following ways:
  - In the Project Explorer view or the EGL Parts List view, right-click on an EGL file that contains a program, library, PageHandler, service, or report handler part. Select **Open in Parts Reference**.
  - Alternatively, open an EGL file in the EGL editor:
    - a. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
    - b. In the Outline view, right-click on a file, then click **Open in Parts Reference**.
2. The program, library, PageHandler, service, or report handler part is at the top level of the hierarchy; each referenced part is a sub-item in that hierarchy; and for each part, the view displays parameters, data declarations, use declarations, and functions, as appropriate.
3. Double-click on a part. The related source file opens in the EGL editor, and the part name is highlighted.

You can search for related EGL code by right-clicking on the part, clicking **References** or **Declarations** and then choosing a scope for the search.

You can also search for parts or text strings among the parts in the view:

1. Right-click on the EGL Parts Reference view and then click **Find in tree**. The EGL Parts Reference Find window opens.
2. Type a search string into the **Search string** field.
  - A question mark (?) represents any one character
  - An asterisk (\*) represents a series of any characters

For example, type *myForm?Group* to locate parts named *myForm1Group* and *myForm2Group*, but not *myForm10Group*. Type *myForm\*Group* to locate parts named *myForm1Group*, *myForm2Group*, and *myForm10Group*.

3. Choose a search type. **Part search** searches for EGL parts and **Text search** searches for text within the
4. Select the options for the search.
  - You can select a type of part to search for by clicking a radio button under **Search For**.
  - Under **Direction**, choose to search **Forward** or **Backward** from the currently selected part.
  - To continue searching from the other end if the search reaches the bottom or the top of the tree, select the **Wrap search** check box.
  - To make a text search case-sensitive (so that *myFormGroup* is different from *MYFORMGROUP*), click the **Case sensitive** check box.
  - To search for only a complete part name or text string, select the **Whole word** check box.
5. When you are finished setting the criteria for the search, click **Find**. The first result of the search is highlighted in the EGL Parts Reference view.
6. To move to the next result, click **Find**.
7. When you are finished, click **Close**.

#### Related concepts

"EGL projects, packages, and files" on page 15

"Parts" on page 19

#### Related tasks

"Locating an EGL source file in the Project Explorer" on page 367

"Opening a part in an .egl file" on page 367

#### Related reference

"EGL editor" on page 577

---

## Viewing lists of parts

You can choose one or more EGL parts and group those parts in a list to filter or sort them. To populate the EGL Parts List view with parts, do one of the following:

- In the Project Explorer view, select one or more EGL resources, such as files, projects, or packages. Then, right-click the selected resources and click **Open in Parts List**.
- In the EGL Parts Reference view, select one or more EGL parts. Then, right-click the selected parts and click **Show in Parts List**.

Once the EGL Parts List view is open and populated with parts, you can do the following things to work with the list of parts:

- Double-click on a part to open it in the EGL editor.
- Click on a column header to sort the list by the part name, type, project, package, or filename.
- Click the refresh button at the top right of the view to refresh the information in the view.
- Open the history drop-down list at the top right of the view to return to a list of parts you have viewed in the EGL Parts List view previously.



- Filter the list of parts with the Filters drop-down list at the top right of the view. For more information, see *Filtering lists of parts*.
- Go to the part in the Project Explorer view by right-clicking it and then clicking **Show in Project Explorer**.
- Generate a generatable part by right-clicking it and then clicking **Generate** or **Generate with wizard**.
- Debug a program by right-clicking it and then clicking **Debug EGL program**.
- Open a part in the EGL Parts Reference view by right-clicking a program, library, pageHandler, or report handler and then clicking **Open in Parts Reference**. See *Viewing part references*.
- Search for related EGL code by right-clicking on the part, clicking **References** or **Declarations** and then choosing a scope for the search.

#### Related concepts

"Parts" on page 19

#### Related tasks

"Filtering lists of parts"

"Searching for parts" on page 363

"Viewing part references" on page 364

"Opening a part in an .egl file" on page 367

---

## Filtering lists of parts

While viewing a list of parts in the EGL Parts List view, you can filter the list to include only certain parts:

1. Populate the EGL Parts List view with EGL parts. For more information, see *Viewing lists of parts*.
2. From the drop-down list at the top right corner of the EGL Parts List view, click **Filters**. The EGL Parts List Filter window opens.
3. In the EGL Parts List Filter window, set the criteria for the filter:
  - To filter by project, folder, package, or file, select or deselect the resources under **Part Resources**.
  - To filter by type of part, select or deselect the types of parts under **Part Type**.
  - To filter by part name, enter a part name in the **Part name filter** field. A question mark (?) represents any one character and an asterisk (\*) represents a series of any characters.
4. When you are finished setting the filter in the EGL Parts List Filter window, click **OK**. Only the parts that match the filter criteria are shown in the EGL Parts List view.

#### Related concepts

"Parts" on page 19

#### Related tasks

"Viewing lists of parts" on page 365

"Searching for parts" on page 363

"Viewing part references" on page 364

"Opening a part in an .egl file" on page 367

---

## Opening a part in an .egl file

With a few keystrokes you can access an EGL part other than a build part, anywhere in your workspace:

1. In the workbench, click **Navigate > Open Part** or click the **Open Part** button on the toolbar. The Open Part dialog is displayed.
2. Type the name of the part you want to locate; or to display a list of parts with names that match a specific pattern of characters, embed wildcard symbols in the name:
  - A question mark (?) represents any one character
  - An asterisk (\*) represents a series of any characters

For example, type *myForm?Group* to locate parts named myForm1Group and myForm2Group, but not myForm10Group. Type *myForm\*Group* to locate parts named myForm1Group, myForm2Group, and myForm10Group.

As you type the name, qualifying parts are displayed in the Open Part dialog, in the Matching parts section.

3. From the list of parts, select the part you want to open. The dialog's Qualifier section displays the path containing the folder, project, package, and source file that holds the selected part. In the event that multiple parts have the same name, select a part by clicking on the path of the file you want to open.
4. Click **OK**. The source file containing the part you selected opens in the EGL editor, with the part name highlighted.

### Related concepts

"EGL projects, packages, and files" on page 15

"Parts" on page 19

### Related tasks

"Creating an EGL source file" on page 130

"Locating an EGL source file in the Project Explorer"

### Related reference

"EGL editor" on page 577

---

## Locating an EGL source file in the Project Explorer

If you are editing an EGL source file, you can quickly locate the file in the Project Explorer view. The Show in Project Explorer context menu option does the following:

- Opens the Project Explorer view, if it is not already open
- Expands the Project Explorer tree nodes needed to locate the source file
- Highlights the source file

To locate an EGL source file in the Project Explorer, do as follows:

1. Right-click within the editor area of an open EGL source file. A context menu displays.
2. Select **Show in Project Explorer** from the context menu.

### Related tasks

"Creating an EGL source file" on page 130

"Opening a part in an .egl file"

**Related reference**  
"EGL editor" on page 577

---

## Deleting an EGL file in the Project Explorer

To delete an EGL file in the Project Explorer, do this:

1. Click the EGL file and press the Delete key. Alternatively, right-click the EGL file and when the context menu is displayed, select **Delete**.
2. You will be asked to confirm that you want to delete the file. Click **Yes** to delete the file or **No** to cancel the deletion.

### **Related tasks**

"Creating an EGL source file" on page 130  
"Locating an EGL source file in the Project Explorer" on page 367

---

## Debugging EGL code

---

### EGL debugger

When you are in the Workbench, the EGL debugger lets you debug EGL code without requiring that you first generate output. These categories are in effect:

- To debug PageHandlers, as well as programs used in a J2EE context, you can use the local WebSphere Application Server test environment in debug mode--
  - You must use that environment for all code that runs under J2EE in a Web application.
  - You may use that environment for programs that run in a batch application under J2EE.
- To debug other code (batch applications that do not run under J2EE; or text applications), use a launch configuration that is outside of the WebSphere Test Environment. In this case, you can start the debug session with a few keystrokes.

If you are working on a batch program that you intend to deploy in a J2EE context, you can use the launch configuration to debug the program in a non-J2EE context. Although your setup is simpler, you need to adjust some values:

- You need to set the value of the build descriptor option J2EE to NO when you use the launch configuration.
- Also, you need to adjust Java property values to conform to differences in accessing a relational database--
  - For J2EE you specify a string like *jdbc/MyDB*, which is the name to which a data source is bound in the JNDI registry. You specify that string in these ways:
    - By setting the build descriptor option *sqlJNDIName*; or
    - By placing a value in the EGL SQL Database Connections preference page, in the Connection JNDI Name field; for details, see *Setting preferences for SQL database connections*.
  - For non-J2EE you specify a connection URL like *jdbc:db2:MyDB*. You specify that string in these ways:
    - By setting the build descriptor option *sqlDB*; or
    - By placing a value in EGL SQL Database Connections preference page, in the field Connection URL; for details, see *Setting preferences for SQL database connections*.

A later section describes the interaction of build descriptors and EGL preferences.

### Debugger commands

You use the following commands to interact with the EGL debugger:

#### Add breakpoint

Identifies a line at which processing pauses. When code execution pauses, you can examine variable values as well as the status of files and screens.

Breakpoints are remembered from one debugging session to the next, unless you remove the breakpoint.

You cannot set a breakpoint at a blank line or at a comment line.

**Disable breakpoint**

Inactivates a breakpoint but does not remove it.

**Enable breakpoint**

Activates a breakpoint that was previously disabled.

**Remove breakpoint**

Clears the breakpoint so that processing no longer automatically pauses at the line.

**Remove all breakpoints**

Clears every breakpoint.

**Run**

Runs the code until the next breakpoint or until the run unit ends. (In any case the debugger stops at the first statement in the main function.)

**Run to line**

Runs all statements up to (but not including) the statement on a specified line.

**Step into**

Runs the next EGL statement and pauses.

The following list indicates what happens if you issue the command **step into** for a particular statement type:

**call**

Stops at the first statement of a called program if the called program runs in the EGL debugger. Stops at the next statement in the current program if the called program runs outside of the EGL debugger.

The EGL debugger searches for the receiving program in every project in the workbench.

**converse**

Waits for user input. That input causes processing to stop at the next running statement, which may be in a validator function.

**forward**

If the code forwards to a PageHandler, the debugger waits for user input and stops at the next running statement, which may be in a validator function.

If the code forwards to a program, the debugger stops at the first statement in that program.

**function invocation**

Stops at the first statement in the function.

**JavaLib.invoke and related functions**

Stops at the next Java statement, so you can debug the Java code that is made available by the Java access functions.

**show, transfer**

Stops at the first statement of the program that receives control. The target program is EGL source that runs in the EGL debugger and is not EGL-generated code.

After either a **show** statement or a **transfer** statement of the form *transfer to a transaction*, the EGL debugger switches to the build descriptor for the new program or (if no such build descriptor is in use) prompts the user for a new build descriptor. The new program can have a different set of properties from the program that ran previously.

The EGL debugger searches for the receiving program in every project in the workbench.

### Step over

Runs the next EGL statement and pauses, but does not stop within functions that are invoked from the current function.

The following list indicates what happens if you issue the command **step over** for a particular statement type:

#### converse

Waits for user input, then skips any validation function (unless a breakpoint is in effect). Stops at the statement that follows the **converse** statement.

#### forward

If the code forwards to a PageHandler, the debugger waits for user input and stops at the next running statement, but not in a validator function, unless a breakpoint is in effect.

If the code forwards to a program, the debugger stops at the first statement in that program.

#### show, transfer

Stops at the first statement of the program that receives control. The target program is EGL source that runs in the EGL debugger and is not EGL-generated code.

After either a **show** statement or a **transfer** statement of the form *transfer to a transaction*, the EGL debugger switches to the build descriptor for the new program or (if no such build descriptor is in use) prompts the user for a new build descriptor. The new program can have a different set of properties from the program that ran previously.

The EGL debugger searches for the receiving program in every project in the workbench.

### Step return

Runs the statements needed to return to an invoking program or function; then, pauses at the statement that receives control in that program or function.

An exception is in effect if you issue the command **step return** in a validator function. In that case, the behavior is identical to that of a **step into** command, which primarily means that the EGL debugger runs the next statement and pauses.

The EGL debugger treats the following EGL statements as if they were null operators:

- **sysLib.audit**
- **sysLib.purge**
- **sysLib.startTransaction**

You can add a breakpoint at these statements, for example, but a **step into** command merely continues to the subsequent statement, with no other effect.

Finally, if you issue the command **step into** or **step over** for a statement that is the last one running in the function (and if that statement is not **return**, **exit program**, or **exit stack**), processing pauses in the function itself so that you can review variables that are local to the function. To continue the debug session in this case, issue another command.

## Use of build descriptors

A build descriptor helps to determine aspects of the debugging environment. The EGL debugger selects the build descriptor in accordance with the following rules:

- If you specified a debug build descriptor for your program or PageHandler, the EGL debugger uses that build descriptor. For details on how to establish the debug build descriptor, see *Setting the default build descriptors*.
- If you did not specify a debug build descriptor, the EGL debugger prompts you to select from a list of your build descriptors or to accept the value **None**. If you accept the value **None**, the EGL debugger constructs a build descriptor for use during the debugging session; and a preference determines whether VisualAge Generator compatibility is in effect.
- If you specified either **None** or a build descriptor that lacks some of the required database-connection information, the EGL debugger gets the connection information by reviewing your preferences. For details on how to set those preferences, see *Setting preferences for SQL database connections*.

If you are debugging a program that is intended for use in a text or batch application in a Java environment, and if that program issues a **transfer** statement that switches control to a program that is also intended for use in a different run unit in a Java environment, the EGL debugger uses a build descriptor that is assigned to the receiving program. The choice of build descriptor is based on the rules described earlier.

If you are debugging a program that is called by another program, the EGL debugger uses the build descriptor that is assigned to the called program. The choice of build descriptor is based on the rules described above, except that if you do not specify a build descriptor, the debugger does not prompt you for a build descriptor when the called program is invoked; instead, the build descriptor for the calling program remains in use.

**Note:** You must use a different build descriptor for the caller and the called program if one of those programs (but not both) takes advantage of VisualAge Generator compatibility. The generation-time status of VisualAge compatibility is determined by the value of build descriptor option **VAGCompatibility**.

A build descriptor or resource association part that you use for debugging code may be different from the one that you use for generating code.

## SQL-database access

To determine the user ID and password to use for accessing an SQL database, the EGL debugger considers the following sources in order until the information is found or every source is considered:

1. The build descriptor used at debug time; specifically, the build descriptor options `sqlID` and `sqlPassword`.
2. The SQL preferences page, as described in *Setting preferences for SQL database connections*; at that page, you also specify other connection information.
3. An interactive dialog that is displayed at connection time. Such a dialog is displayed only if you select the checkbox **Prompt for SQL user ID and password when needed**.

The user ID and password used to access an SQL database are separate from the user ID and password used to make remote calls while debugging. To set the user ID and password for remote calls while debugging, see *Setting preferences for the EGL debugger*.

## call statement

As noted earlier, the EGL debugger responds to a **transfer** or **show** statement by interpreting EGL source code. The EGL debugger responds to a **call** statement, however, by reviewing the linkage options part specified in the build descriptor, if any. If the referenced linkage options part includes a **callLink** element for the call, the result is as follows:

- If the **callLink** property **remoteComType** is set to **DEBUG**, the EGL debugger interprets EGL source code. The debugger finds the source by referencing the **callLink** properties **package** and **location**.
- If the **callLink** property **remoteComType** is not set to **DEBUG**, the debugger invokes EGL-generated code and uses the information in the linkage options part as if the debugger were running an EGL-generated Java program.

In the absence of linkage information, the EGL debugger responds to a **call** statement by interpreting EGL source code. Linkage information is unavailable in these cases:

- No build descriptor is used; or
- A build descriptor is used, but no linkage options part is specified in that build descriptor; or
- A linkage options part is specified in the build descriptor, but the referenced part does not have a **callLink** element that references the called program.

If the debugger runs EGL source code, you can run statements in that program by issuing the **step into** command from the caller. If the debugger calls generated code, however, the debugger runs the entire program; the **step into** command works like the **step over** command.

## System type used at debug time

A value for system type is available in `sysVar.systemType`. Also, a second value is available in `VGLib.getVAGSysType` if you requested development-time compatibility with VisualAge Generator).

The value in `sysLib.systemType` is the same as the value of the build descriptor option **system**, except that the value is **DEBUG** in either of two cases:

- You select the preference **Set systemType to DEBUG**, as mentioned in *Setting preferences for the EGL debugger*; or
- You specified **NONE** as the build descriptor to use during the debugging session, regardless of the value of that preference.

The system function `VGLib.getVAGSysType` returns the VisualAge Generator equivalent of the value in `sysLib.systemType`; for details, see the table in `VGLib.getVAGSysType`.

## EGL debugger port

The EGL debugger uses a port to establish communication with the Eclipse workbench. The default port number is 8345. If another application is using that port or if that port is blocked by a firewall, set a different value as described in *Setting preferences for the EGL debugger*.



If a value other than 8345 is specified as the EGL debugger port and if an EGL program will be debugged on the J2EE server, you must edit the server configuration:

1. Go to the Environment tab, System Properties section
2. Click Add
3. For Name, type `com.ibm.debug.egl.port`
4. For Value, type the port number

## Recommendations

As you prepare to work with the EGL debugger, consider these recommendations (most of which assume that `sysVar.systemType` is set to `DEBUG` when you are debugging the code):

- If you are retrieving a date from a database but expect the runtime code to retrieve that date in a format other than the ISO format, write a function to convert the date, but invoke the function only when the system type is `DEBUG`. The ISO format is `yyyy-mm-dd`, which is the only one available to the debugger.
- To specify external Java classes for use when the debugger runs, modify the class path, as described in *Setting preferences for the EGL debugger*. You might need extra classes, for example, to support MQSeries, JDBC drivers, or Java access functions.
- When you are debugging a PageHandler function that was invoked by JSF (rather than by another EGL function), use Run to leave the function rather than Step Over, Step Into, or Step Return. Using any of the three Step commands takes you to the generated Java code of the PageHandler, which is not useful when you are debugging EGL. If you use one of the Step commands, use Run to leave the generated Java code and display the Web page in a browser.
- If you are using the SQL option `WITH HOLD` (or the EGL equivalent), you need to know that the option `WITH HOLD` is unavailable for EGL-generated Java or in the EGL debugger. You may be able to work around the limitation, in part by placing commit statements inside a conditional statement that is invoked only at run time, as in the following example:

```
if (systemType not debug)
  sysLib.commit();
end
```

If EGL programs are debugged on the J2EE server or by an EGL Listener, the server or EGL Listener must be configured to indicate the number for the EGL debugger port:

- To configure a J2EE server, edit the server configuration--
  1. Go to the Environment tab, System Properties section
  2. Click **Add**
  3. For **Name**, type `com.ibm.debug.egl.port`
  4. For **Value**, type the new port number
- To configure an EGL Listener, edit the EGL Listener launch configuration--
  1. Go to the Arguments tab
  2. In the **VM Arguments** field, type this:

```
-Dcom.ibm.debug.egl.port=portNumber
```

*portNumber*

The new port number

### Related concepts

"Compatibility with VisualAge Generator" on page 532  
"Character encoding options for the EGL debugger" on page 117  
"VSAM support" on page 335

### Related tasks

"Setting preferences for SQL database connections" on page 121  
"Setting preferences for the EGL debugger" on page 116  
"Setting the default build descriptors" on page 118

### Related reference

"remoteComType in callLink element" on page 511  
"sqlDB" on page 490  
"sqlID" on page 491  
"sqlJNDIName" on page 492  
"sqlPassword" on page 492  
"getVAGSysType()" on page 1054  
"systemType" on page 1074

---

## Debugging applications other than J2EE

### Starting a non-J2EE application in the EGL debugger

To start debugging an EGL text program or non-J2EE basic program in an EGL debugging session, a launch configuration is required. A launch configuration defines a program's file location and specifies how the program should be launched. You can let the EGL application create the launch configuration (implicit creation), or you can create one yourself (see *Creating a launch configuration in the EGL debugger*).

To launch a program using an implicitly created launch configuration, do as follows:

1. In the Project Explorer view, right-click the EGL source file you want to launch. Alternatively, if the EGL source file is open in the EGL editor, you can right-click on the program in the Outline view.
2. A context menu displays.
3. Click **Debug EGL Program**. A launch configuration is created, and the program is launched in the EGL debugger.

To view the implicitly created launch configuration, do as follows:

1. Click the arrow next to the Debug button on the toolbar. A context menu displays.
2. Click **Debug**. The Debug dialog displays. The name of the launch configuration is displayed in the Name field. Implicitly created launch configurations are named according to the project and source file names.

**Note:** You can also display the Debug dialog by clicking **Debug** from the Run menu.

### Related concepts

"EGL debugger" on page 369

### Related tasks

"Creating a launch configuration in the EGL debugger" on page 376  
"Stepping through an application in the EGL debugger" on page 380

“Using breakpoints in the EGL debugger” on page 379

“Viewing variables in the EGL debugger” on page 380

## Creating a launch configuration in the EGL debugger

To start debugging an EGL text program or non-J2EE basic program in an EGL debugging session, a launch configuration is required. A launch configuration defines how a program should be launched. You can create a launch configuration (explicit creation), or you can let the EGL application create one for you (see *Starting a non-J2EE program in the EGL debugger*).

To start a program using an explicitly created launch configuration, do as follows:

1. Click the arrow next to the Debug button on the toolbar, then click **Debug**, or select **Debug** from the Run menu.
2. The Debug dialog is displayed.
3. Click **EGL Program** in the Configurations list, then click **New**.
4. If you did not have an EGL source file highlighted in the Project Explorer view, the launch configuration is named *New\_configuration*. If you had an EGL source file highlighted in the Project Explorer view, the launch configuration has the same name as the EGL source file. If you want to change the name of the launch configuration, type the new name in the Name field.
5. If the name in the Project field of the Load tab is not correct, click **Browse**. A list of projects displays. Click a project, then click **OK**.
6. If the name in the EGL program source file field is not correct or the field is empty, click **Search**. A list of EGL source files displays. Click a source file, then click **OK**.
7. If you made changes to any of the fields on the Debug dialog, click **Apply** to save the launch configuration settings.
8. Click **Debug** to launch the program in the EGL debugger.

**Note:** If you have not yet used **Apply** to save the launch configuration settings, clicking **Revert** will remove all changes that you have made.

### Related concepts

“EGL debugger” on page 369

### Related tasks

“Starting a non-J2EE application in the EGL debugger” on page 375

“Stepping through an application in the EGL debugger” on page 380

“Using breakpoints in the EGL debugger” on page 379

“Viewing variables in the EGL debugger” on page 380

## Creating an EGL Listener launch configuration

To debug a non-J2EE EGL application called from an EGL-generated Java application or wrapper, an EGL Listener launch configuration is required. To create an EGL Listener launch configuration, do as follows:

1. Click the arrow next to the Debug button on the toolbar, then click **Debug**, or select **Debug** from the Run menu.
2. The Debug dialog is displayed.
3. Click **EGL Listener** in the Configurations list, then click **New**.
4. The Listener launch configuration is named *New\_configuration*. If you want to change the name of the launch configuration, type the new name in the Name field.

5. If you do not enter a port number, the port defaults to 8346; otherwise, enter a port number. Each EGL Listener requires its own port.
6. Click **Apply** to save the Listener launch configuration.
7. Click **Debug** to launch the EGL Listener.

#### Related concepts

"EGL debugger" on page 369

#### Related tasks

"Creating a launch configuration in the EGL debugger" on page 376

"Starting a non-J2EE application in the EGL debugger" on page 375

"Stepping through an application in the EGL debugger" on page 380

"Using breakpoints in the EGL debugger" on page 379

"Viewing variables in the EGL debugger" on page 380

---

## Debugging J2EE applications

### Preparing a server for EGL Web debugging

To debug EGL Web programs that run in the WebSphere Application Server, you must prepare the server for debugging. The preparation step must be done once per server and does not need to be done again, even if the Workbench is shut down.

To prepare a server for debugging, do as follows:

1. If you are working with WebSphere v5.1 Test Environment, make sure that the server is stopped. If you are working with WebSphere Application Server v6.0, make sure that the server is running. The explanation for this difference is that the v6.0 code is a functioning server.
2. In the Server view, right-click on the server. A context menu displays.
3. Select **Enable/Disable EGL Debugging**. A message indicates that you have enabled EGL debugging.
4. If you want to debug the generated Java instead of EGL, right-click on the server again and select **Enable/Disable EGL Debugging**. A message indicates that you have disabled EGL debugging.

#### Related concepts

"EGL debugger" on page 369

"WebSphere Application Server and EGL" on page 425 "Web support" on page 215

#### Related tasks

"Starting an EGL Web debugging session" on page 378

"Starting a server for EGL Web debugging"

"Stepping through an application in the EGL debugger" on page 380

"Using breakpoints in the EGL debugger" on page 379

"Viewing variables in the EGL debugger" on page 380

### Starting a server for EGL Web debugging

If you are working with an EGL-based Web application that accesses a JNDI data source, you cannot follow the instructions in the current topic unless you previously configured a Web application server. For background information that is specific to WebSphere, see *WebSphere Application Server and EGL*.

Also, if you wish to debug an EGL Web program, you must prepare the server for that purpose as described in *Preparing a server for EGL Web debugging*.

To start the server for debugging, do as follows:

1. In the Server view, right-click on the server
2. Select **Debug > Debug on Server**

#### Related concepts

"EGL debugger" on page 369

"WebSphere Application Server and EGL" on page 425 "Web support" on page 215

#### Related tasks

"Preparing a server for EGL Web debugging" on page 377

"Starting an EGL Web debugging session"

"Stepping through an application in the EGL debugger" on page 380

"Using breakpoints in the EGL debugger" on page 379

"Viewing variables in the EGL debugger" on page 380

## Starting an EGL Web debugging session

If you are working with an EGL-based Web application that accesses a JNDI data source, you cannot follow the instructions in the current topic unless you previously configured a Web application server. For background information that is specific to WebSphere, see *WebSphere Application Server and EGL*.

Also, if you wish to debug an EGL Web program, you must prepare the server for that purpose as described in *Preparing a server for EGL Web debugging*. You will save time on the current procedure if you already started the server for debugging, as described in *Starting a server for EGL Web debugging*.

To start an EGL Web debugging session, do as follows:

1. In the Project Explorer, expand the **WebContent** and **WEB-INF** folders. Right-click the JSP file you want to run, then select **Debug > Debug on Server**. The Server Selection dialog is displayed.
2. If you have already configured a server for this Web project, select **Choose an existing server**, then select a server from the list. Click **Finish** to start the server (if necessary), to deploy the application to the server, and to start the application.
3. If you have not configured a server for this Web project, you can proceed as follows, but only if your application does not access a JNDI data source--
  - a. Select **Manually define a server**.
  - b. Specify the host name, which (for the local machine) is **localhost**.
  - c. Select a server type that is similar to the Web application server on which you intend to deploy your application at run time. Choices include **WebSphere v5.1 Test Environment** and **WebSphere v6.0 Server**.
  - d. If you do not intend to change your choices as you work on the current project, select the check box for **Set server as project default**.
  - e. In most cases, you can avoid this step; but if you wish to specify settings that are different from the defaults, click **Next** and make your selections.
  - f. Click **Finish** to start the server, to deploy the application to the server, and to start the application.

### Related concepts

"EGL debugger" on page 369

"WebSphere Application Server and EGL" on page 425 "Web support" on page 215

### Related tasks

"Preparing a server for EGL Web debugging" on page 377

"Starting a server for EGL Web debugging" on page 377

"Stepping through an application in the EGL debugger" on page 380

"Using breakpoints in the EGL debugger"

"Viewing variables in the EGL debugger" on page 380

---

## Using breakpoints in the EGL debugger

Breakpoints are used to pause execution of a program. You can manage breakpoints inside or outside of an EGL debugging session. Keep the following in mind when working with breakpoints:

- A blue marker in the left margin of the Source view indicates that a breakpoint is set and enabled.
- A white marker in the left margin of the Source view indicates that a breakpoint is set but disabled.
- The absence of a marker in the left margin indicates that a breakpoint is not set.

### Add or remove a breakpoint

Add or remove a single breakpoint in an EGL source file by doing one of the following:

- Position the cursor at the breakpoint line in the left margin of the Source view and double-click.
- Position the cursor at the breakpoint line in the left margin of the Source view and right-click. A context menu displays. Click the appropriate menu item.

### Disable or enable a breakpoint

Disable or enable a single breakpoint in an EGL source file by doing the following:

1. In the Breakpoint view, right-click on the breakpoint. A context menu displays.
2. Click the appropriate menu item.

### Remove all breakpoints

Remove all breakpoints from an EGL source file by doing the following:

1. Right-click on any of the breakpoints displayed in the Breakpoints view. A context menu displays.
2. Click **Remove All**.

### Related concepts

"EGL debugger" on page 369

### Related tasks

"Creating a launch configuration in the EGL debugger" on page 376

"Starting a non-J2EE application in the EGL debugger" on page 375

"Stepping through an application in the EGL debugger" on page 380

"Viewing variables in the EGL debugger" on page 380

---

## Stepping through an application in the EGL debugger

As explained in *EGL debugger*, the EGL debugger provides the following commands to control execution of a program during a debugging session:

### Resume

Runs the code until the next breakpoint or until the end of the program.

### Run to Line

Allows you to select an executable line in the Source view and run the code to that line.

### Step Into

Runs the next EGL statement and pauses. The program stops at the first statement of a called function.

### Step Over

Runs the next EGL statement and pauses, but does not stop within functions that are invoked from the current function.

### Step Return

Returns to an invoking program or function.

With the exception of Run to Line, each of the commands can be accessed in the following ways:

- Click the appropriate button on the toolbar of the Debug view; or
- Click the appropriate menu item on the Run menu; or
- Right-click a highlighted thread in the Debug view, then click the appropriate menu item.

To use Run to Line, do as follows when the program is paused:

1. Position the cursor in the left margin of the Source view at an executable line, then right-click. A context menu displays.
2. Click **Run to Line**.

When using Run to Line, keep in mind the following:

- The operation is not available from the Debug view or the Run menu
- Run to Line stops at enabled breakpoints

### Related concepts

“EGL debugger” on page 369

### Related tasks

“Creating a launch configuration in the EGL debugger” on page 376

“Starting a non-J2EE application in the EGL debugger” on page 375

“Using breakpoints in the EGL debugger” on page 379

“Viewing variables in the EGL debugger”

---

## Viewing variables in the EGL debugger

Whenever a program is paused, you can view the current values of the program’s variables.

To view a program’s variables, do as follows:

1. In the Variables view, expand the parts in the navigator to see their variables.

2. To display the variables' types, click the **Show Type Names** button on the toolbar.
3. To display the details of a variable in a separate pane, click on the variable, then click the **Show Detail** button on the toolbar.

**Related concepts**

"EGL debugger" on page 369

**Related tasks**

"Creating a launch configuration in the EGL debugger" on page 376

"Starting a non-J2EE application in the EGL debugger" on page 375

"Stepping through an application in the EGL debugger" on page 380

"Using breakpoints in the EGL debugger" on page 379





---

## Working with EGL build parts

---

### Creating a build file

To create a build file, do as follows:

1. Identify a project or folder to contain the file. You must create a project or folder if you do not already have one. The project should be an EGL or EGL Web project.
2. In the workbench, click **File > New > EGL Build File**.
3. Select the project or folder that will contain the EGL build file. In the File name field, type the name of the EGL build file, for example MyEGLbuildParts. The extension .eglbld is required for the file name. An extension is automatically appended to the end of the file name if no extension or an invalid extension is specified.
4. Click **Finish** to create the build file with no EGL build part declaration. The build file appears in the Project Explorer view and automatically opens in the EGL build parts editor.
5. To add an EGL build part before creating the build file, click **Next**. Select the type of build part to add, then click **Next**. Type a name and a description for the build part, then click **Finish**. The build file appears in the Project Explorer view and automatically opens in the EGL build parts editor.

#### Related concepts

"EGL projects, packages, and files" on page 15

"Introduction to EGL" on page 1

#### Related tasks

"Adding a build descriptor part to an EGL build file" on page 387

"Adding a linkage options part to an EGL build file" on page 401

"Adding an import statement to an EGL build file" on page 406

"Adding a resource associations part to an EGL build file" on page 397

"Creating an EGL Web project" on page 127

## Setting up general build options

### Build descriptor part

A build descriptor part controls the generation process. The part contains several kinds of information:

- *Build descriptor options* specify how to generate and prepare EGL output, and a subset of the build descriptor options can cause other build parts to be included in the generation process. For details on specific options, see *Build descriptor options*.
- *Java runtime properties* assign values to the following properties:
  - `vgj.datemask.gregorian.long.locale`, which contains the date mask used in either of two cases:
    - The Java code generated for the system variable `VGVar.currentFormattedGregorianCalendar` is invoked; or

- EGL validates a page item or text-form field that has a length of 10 or more, if the item property **dateFormat** is set to *systemGregorianCalendarFormat*. The meaning of *locale* is described at the end of this section.
  - *vgj.datemask.gregorian.short.locale*, which contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property **dateFormat** is set to *systemGregorianCalendarFormat*. The meaning of *locale* is described at the end of this section.
  - *vgj.datemask.julian.long.locale*, which contains the date mask used in either of two cases:
    - The Java code generated for the system variable *VGVar.currentFormattedJulianDate* is invoked; or
    - EGL validates a page item or text-form field that has a length of 8 or more, if the item property **dateFormat** is set to *systemJulianDateFormat*. The meaning of *locale* is described at the end of this section.
  - *vgj.datemask.julian.short.locale*, which contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property **dateFormat** is set to *systemJulianDateFormat*. The meaning of *locale* is described at the end of this section.
  - *vgj.jdbc.database.SN*, which identifies a database that is made available to your Java code.
 

You must customize the name of the property itself when you specify a substitution value for *SN*, at deployment time. The substitution value in turn must match either the server name that is included in the invocation of *VGLib.connectionService* or the database name that is included in the invocation of *sysLib.connect*.
- You also must customize the name of the date-mask properties:
- In a given run unit, each property that is initially in effect has a name whose last qualifier (the *locale*) matches the value in the program property **vgj.nls.code**
  - In a Web application, a different set of properties is in effect if a program sets the system variable *sysLib.setLocale*

**Master build descriptors:** Your system administrator may require that you use a *master build descriptor* to specify information that cannot be overridden and that is in effect for every generation that occurs in your installation of EGL. By a mechanism described in Master build descriptor, the system administrator identifies that part by name, along with the EGL build file that contains the part.

If the information in the master build descriptor is not sufficient for a particular generation process or if no master build descriptor is identified, you can specify a build descriptor at generation time, along with the EGL build file that contains the generation-specific part. The generation-specific build descriptor (like the master build descriptor) must be at the top level of an EGL build file.

You can create a chain of build descriptors from the generation-specific build descriptor, so that the first in the chain is processed before the second, and the second before the third. When you define a given build descriptor, you begin a chain (or continue one) by assigning a value to the build descriptor option **nextBuildDescriptor**. Your system administrator can use the same technique to create a chain from the master build descriptor. The implication of chaining information is described later.

Any build part referenced by a build descriptor must be visible to the referencing build descriptor, in accordance with the rules described in References to parts. The build part can be a linkage options part or a resource associations part, for example, or the next build descriptor.

**Precedence of options:** For a given build descriptor option (or Java runtime property), the value that is initially processed at generation time stays in effect, and the overall order of precedence is as follows:

1. The master build descriptor
2. The generation-specific build descriptor, followed by the chain that extends from it
3. The chain that extends from the master build descriptor

The benefit of this scheme is convenience:

- The system administrator can specify unchanging values by setting up a master build descriptor.
- You can use a generation-specific build descriptor to assign values that are specific to a generation.
- A project manager can specify a set of defaults by customizing one or more build descriptors. In most situations of this kind, the generation-specific build descriptor points to the first build descriptor in a chain that was developed by the project manager.

Default options can be useful when your organization develops a set of programs that must be generated or prepared similarly.

- The system administrator can create a set of general defaults by establishing a chain that extends from the master build descriptor, although use of this feature is unusual.

If a given build descriptor is used more than once, only the first access of that build descriptor has an effect. Also, only the first specification of a particular option has an effect.

**Example:** Let's assume that the master build descriptor contains these (unrealistic) option-and-value pairs:

OptionX	02
OptionY	05

In this example, the generation-specific build descriptor (called myGen) contains these option-and-value pairs.

OptionA	20
OptionB	30
OptionC	40
OptionX	50

As identified in myGen, the next build descriptor is myNext01, which contains these:

OptionA	120
OptionD	150

As identified in myNext01, the next build descriptor is myNext02, which contains these:

OptionB	220
OptionD	260
OptionE	270

As identified in the master build descriptor, the next build descriptor is myNext99, which contains this:

OptionZ	99
---------	----

EGL accepts option values in the following order:

1. Values for options in the master build descriptor:

OptionX	02
OptionY	05

Those options override all others.

2. Values in the generation-specific build descriptor myGen:

OptionA	20
OptionB	30
OptionC	40

The value for optionX in myGen was ignored.

3. Values for other options in myNext01 and myNext02:

OptionD	150
OptionE	270

The value for optionA in myNext01 was ignored, as was the value for optionD in myNext02.

4. Values for other options in myNext99:

OptionZ	99
---------	----

### Related concepts

"Build" on page 411

"Java runtime properties" on page 431

"References to parts" on page 23

"Master build descriptor"

"Parts" on page 19

### Related tasks

"Adding a build descriptor part to an EGL build file" on page 387

"Adding a resource associations part to an EGL build file" on page 397

"Editing general options in a build descriptor" on page 388

"Editing Java runtime properties in a build descriptor" on page 391

"Editing a resource associations part in an EGL build file" on page 397

### Related reference

"Build descriptor options" on page 464

"Java runtime properties (details)" on page 642

"connect()" on page 1025

"connectionService()" on page 1047

"setLocale()" on page 1038

"currentFormattedGregorianCalendar" on page 1078

"currentFormattedJulianDate" on page 1079

## Master build descriptor

An installation can provide its own set of default values for build options and control whether those default values can be overridden.

To set up the master build descriptor, create two build descriptor parts in the same build file, the first referencing the second by use of the build descriptor option

**nextBuildDescriptor.** The options in the first part specify default values for options that may not be overridden. The options in the second part specify default values for options that can be overridden.

To install the master build descriptor in the workbench, add a plugin.xml file like following to the workbench plugins directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="egl.master.build.descriptor.plugin"
  name="EGL Master Build Descriptor Plug-in"
  version="5.0"
  vendor-name="IBM">
  <requires />
  <runtime />
  <extension point =
    "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor">
    <masterBuildDescriptor
      file = "filePath.buildFileName"
      name = "masterBuildPartName" />
    </extension>
  </plugin>
```

The file path (*filePath*) is in relation to the workspace directory.

If you are using the EGL SDK, you declare the name and file path name of the master build descriptor in a file named `eglmaster.properties`. This file must be in a directory that is listed in the CLASSPATH environment variable. The format of the properties file is as follows:

```
masterBuildDescriptorName=masterBuildPartName
masterBuildDescriptorFile=fullyQualifiedPathforEGLBuildFile
```

#### Related concepts

“Build” on page 411  
“Build descriptor part” on page 383  
“Build plan” on page 413  
“EGL projects, packages, and files” on page 15

#### Related tasks

“Adding a build descriptor part to an EGL build file”

#### Related reference

“Build descriptor options” on page 464  
“Format of `eglmaster.properties` file” on page 585  
“Format of master build descriptor `plugin.xml` file” on page 602

### Adding a build descriptor part to an EGL build file

A build descriptor part controls the generation process. It contains option names and their related values, and those option-and-value pairs specify how to generate and prepare EGL output. Some options specify other control parts, such as a resource association part, that are in the generation process. You can add a build descriptor part to an EGL build file. See *Build descriptor part* for more information. To add a build descriptor part, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**.

2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
3. In the Outline view, right-click on the build file, then click **Add Part**.
4. Click the **Build Descriptor** radio button, then click **Next**.
5. Choose a name for your build descriptor that adheres to EGL part name conventions. In the Name field, type the name of your build descriptor.
6. In the Description field, type a description of your build part.
7. Click **Finish**. The build descriptor is declared in the EGL build file and the build descriptor general options are displayed in the EGL build parts editor.
8. You can optionally create a chain of build descriptors, so that the first in the chain is processed before the second, and the second before the third. If you want to begin or continue a chain of build descriptors, specify the next build descriptor in the **nextBuildDescriptor** option field of the Options list. To populate the **nextBuildDescriptor** option field, do as follows:
  - a. Using the scroll bar on the Options list, scroll down until the **nextBuildDescriptor** option is in view.
  - b. If the nextBuildDescriptor row is not highlighted, click once to select the row.
  - c. Click the Value field once to put the field into edit mode.
  - d. You can type the name of the next build descriptor in the Value field or select an existing build descriptor from the drop-down list.

#### Related concepts

"Build descriptor part" on page 383

#### Related tasks

"Editing general options in a build descriptor"

"Editing Java runtime properties in a build descriptor" on page 391

"Removing a build descriptor part from an EGL build file" on page 392

#### Related reference

"EGL build-file format" on page 462

"Naming conventions" on page 778

### Editing general options in a build descriptor

A build descriptor part controls the generation process. To edit the general build descriptor options and symbolic parameters, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
3. In the Outline view, right-click on a build descriptor and select **Open**. There are three buttons in the upper right corner of the editor view. Make sure that the Show General Build Descriptor Options button (the button on the left) is pressed. The EGL build parts editor displays the general build descriptor options for the current part definition.
4. You can optionally create a chain of build descriptors, so that the first in the chain is processed before the second, and the second before the third. If you want to begin or continue a chain of build descriptors, specify the next build descriptor in the **nextBuildDescriptor** field. If the nextBuildDescriptor row is

not highlighted, click once to select the row, then click the Value field once to put the field into edit mode. You can type the name of the next build descriptor in the Value field or select an existing build descriptor from the drop-down list.

5. To specify the generation and preparation of EGL output, select a grouping of option-and-value pairs from the **Build option filter** drop-down list. If the option you want to define is not highlighted, click once to select the row, then click the Value field once to put the field into edit mode. You can type the option value, or if a drop-down list is available, select an existing value. If you want to limit your view of option-and-value pairs to the ones you have defined, click the Show only options specified check box.

#### Related concepts

"Build descriptor part" on page 383

#### Related tasks

"Adding a build descriptor part to an EGL build file" on page 387

"Editing Java runtime properties in a build descriptor" on page 391

"Removing a build descriptor part from an EGL build file" on page 392

#### Related reference

"EGL build-file format" on page 462

### Choosing options for Java generation

Build descriptor options are set in build descriptor parts. To choose build descriptor options, start the EGL editor and edit the build descriptor part.

When you begin editing a build descriptor part from the GUI, the EGL editor contains a pane listing all EGL build descriptor options. To limit the display to options that are applicable to a program, select a category from the Build option filter drop-down menu.

Select each option you want, and set its value. The value can be literal, symbolic, or a combination of literal and symbolic. You can define symbolic parameters in the EGL part editor; for details, see *Editing general build descriptor options*.

Two build descriptor options—**genDirectory** and **destDirectory**—let you use a symbolic parameter for the value or a portion of the value. For example, for the value of **genDirectory** you can specify `C:\genout\%EZEENV%`. Then if you generate for a Windows environment, the actual generation directory is `C:\genout\WIN`.

You do not need to specify all the options listed. If you do not specify a value for a build descriptor option, the default for the option is used when the option is applicable in the generation context.

If you have specified a master build descriptor, the option values in that build descriptor override the values in all other build descriptors. When you generate, the master and generation build descriptors can chain to other build descriptors as described in *Build descriptor part*.

#### Related concepts

"Build descriptor part" on page 383

#### Related tasks

"Editing general options in a build descriptor" on page 388

"Generating Java wrappers" on page 390



### Related reference

“Build descriptor options” on page 464

## Generating Java wrappers

You can generate Java wrapper classes when you generate the related program. For details on how to set up the build descriptor, see *Java wrapper*.

### Related concepts

“Generation” on page 409

“Generation of Java code into a project” on page 409 “Java wrapper”

### Related tasks

“Building EGL output” on page 413

“Processing Java code that is generated into a directory” on page 420

### Related reference

“Build descriptor options” on page 464

“Java wrapper classes” on page 652

“Output of Java wrapper generation” on page 782

**Java wrapper:** A Java wrapper is a set of classes that act as an interface between the following executables:

- A servlet or a hand-written Java program, on the one hand
- A generated program or EJB session bean, on the other

You generate the Java wrapper classes if you use a build descriptor that has these characteristics:

- The build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**; and
- The build descriptor option **linkage** references a linkage options part that includes a **callLink** element to guide the call from wrapper to program; and
- One of two statements apply:
  - The call from wrapper to program is by way of an EJB session bean (in which case the **callLink** element, **linkType** property is set to **ejbCall**); or
  - The call from wrapper to program is remote (in which case the **callLink** element, **type** property is set to **remoteCall**); also, the **callLink** element, **javaWrapper** property is set to **yes**.

If an EJB session bean mediates between the Java wrapper classes and an EGL-generated program, you generate the EJB session if you use a build descriptor that has these characteristics:

- The build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**; and
- The build descriptor option **linkage** references a linkage options part that includes a **callLink** element to guide the call from wrapper to EJB session bean (in which case the **type** property of the **callLink** element is set to **ejbCall**).

For further details on using the classes, see *Java wrapper classes*. For details on the class names, see *Generated output (reference)*.

### Related concepts

“Generated output” on page 625

“Java program, PageHandler, and library” on page 414

“Runtime configurations” on page 11

### Related tasks

“Generating Java wrappers” on page 390

### Related reference

“Generated output (reference)” on page 626

“Java wrapper classes” on page 652

“Output of Java wrapper generation” on page 782

## Editing Java runtime properties in a build descriptor

When you are editing a build descriptor part, you can assign values to the following Java runtime properties, which are detailed in *Java runtime properties (details)*:

- `vgj.jdbc.database.SN`
- `vgj.datemask.gregorian.long.locale`
- `vgj.datemask.gregorian.short.locale`
- `vgj.datemask.julian.long.locale`
- `vgj.datemask.julian.short.locale`

Your assignments are used only if you generate Java code.

To edit the properties, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
3. In the Outline view, right-click on a build descriptor and select **Open**. The EGL part editor displays the general build descriptor options for the current part definition.
4. Click the **Show Java runtime Properties** button on the editor toolbar.
5. To add the Java runtime property `vgj.jdbc.database.SN`, do this:
  - a. In the screen area that is titled “Database mappings for connect”, click the **Add** button
  - b. Type a “Server name” that you use when coding the system word `VGLib.connectionService`; this value is substituted for `SN` in the name of the generated property
  - c. If the row in the Database mappings for connect list is not highlighted, click once to select the row, then click the JNDI name or URL field once to put the field into edit mode. Type a value whose meaning is different for J2EE connections as compared with non-J2EE connections:
    - In relation to J2EE connections (as is needed in a production environment), the value is the name to which the datasource is bound in the JNDI registry; for example, `jdbc/MyDB`
    - In relation to a standard JDBC connection (as may be used for debugging), the value is the connection URL; for example, `jdbc:db2:MyDB`
6. To assign the date masks used when you code either `VGVar.currentFormattedGregorianCalendar` (for a Gregorian date) or `VGVar.currentFormattedJulianDate` (for a Julian date); or EGL validates a page item or a text-form field that has a length of 10 or more and a **dateFormat** property of `systemGregorianCalendarFormat` or `systemJulianDateFormat`, do this:

- a. In the screen area that is titled "Date Masks", click the **Add** button
  - b. In the Locale column, select one of the codes in the listbox; the selected value is substituted for *locale* in the date-mask properties listed earlier. Only one of your entries is used at run time: the entry for which the value of *locale* matches the value of the Java runtime property **vgj.nls.code**
  - c. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Long Gregorian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
  - d. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Long Julian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
7. To assign the date masks used when EGL validates a page item or a text-form field that has a length less than 10 and a **dateFormat** property of *systemGregorianCalendar* or *systemJulianDate*, do this:
    - a. In the screen area that is titled "Date Masks", click the **Add** button
    - b. In the Locale column, select one of the codes in the listbox; the selected value is substituted for *locale* in the date-mask properties listed earlier. Only one of your entries is used at run time: the entry for which the value of *locale* matches the value of the Java runtime property **vgj.nls.code**
    - c. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Short Gregorian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
    - d. If the row in the Date Masks list is not highlighted, click once to select the row, then click the Short Julian Mask field once to put the field into edit mode. Either select a mask from the listbox or type a mask; characters other than D, Y, or digits can be used as separators, and the default value is specific to the locale
  8. To remove an assignment, click on it, then click the **Remove** button.

#### Related concepts

"Build descriptor part" on page 383  
 "Java runtime properties" on page 431

#### Related tasks

"Adding a build descriptor part to an EGL build file" on page 387  
 "Editing general options in a build descriptor" on page 388  
 "Removing a build descriptor part from an EGL build file"

#### Related reference

"EGL build-file format" on page 462  
 "Java runtime properties (details)" on page 642  
 "connectionService()" on page 1047  
 "currentFormattedGregorianCalendar" on page 1078  
 "currentFormattedJulianDate" on page 1079

### Removing a build descriptor part from an EGL build file

To remove a build descriptor part from an EGL build file, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu
3. In the Outline view, right-click on the build descriptor part, then click **Remove**

#### Related concepts

"Build descriptor part" on page 383

#### Related tasks

"Adding a build descriptor part to an EGL build file" on page 387

"Editing general options in a build descriptor" on page 388

"Editing Java runtime properties in a build descriptor" on page 391

## Setting up external file, printer, and queue associations

### Resource associations and file types

An EGL fixed record that accesses an external file, printer, or queue has a logical file or queue name. (In the case of a printer, the logical file name is *printer* for most runtime systems.) The name can be no more than 8 characters and is meaningful only as a way of relating the record to a *system name*, which the target system uses to access a physical file, printer, or queue.

In relation to files or queues, the file or queue name is a default for the system name. In relation to printers, no default exists.

Instead of accepting a default, you can take one or both of these actions:

- At generation time, you control the generation process with a build descriptor that in turn references a specific resource associations part. The resource associations part relates the file name with a system name on the target platform where you intend to deploy the generated code.
- At run time (in most cases) you can change the value in the record-specific variable `resourceAssociation` (for files or queues) or in the system variable `ConverseVar.printerAssociation` (for print output). Your purpose is to override the system name that you specified either by default or by specifying a resource associations part.

The resource associations part does not apply to these record types:

- `basicRecord`, because basic records do not interact with data stores
- `SQLRecord`, because SQL records interact with relational databases
- `DLISegment`, because DL/I segment records interact with hierarchical databases

**Resource associations part:** The resource associations part is a set of association elements, each of which has these characteristics:

- Is specific to a logical file or queue name
- Has a set of entries, each specific to a target system; each entry identifies the file type on the target platform, along with the system name and in some cases additional information

You can think of an association element as a set of properties and values in a hierarchical relationship, as in the following example:

```

// an association element
property: fileName
value:    myFile01

// an entry, with multiple properties
property: system
value:    aix
property: fileType
value:    spool
property: systemName
value:    employee

// a second entry
property: system
value:    win
property: fileType
value:    seqws
property: systemName
value:    c:\myProduct\myFile.txt

```

In this example, the file name `myFile01` is related to these files:

- *employee* on AIX
- *myFile.txt* on Windows 2000/NT/XP

The file name must be a valid name, an asterisk, or the beginning of a valid name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names. An association element that includes the following value for a file name, for example, pertains to any file name that begins with the letters *myFile*:

```
myFile*
```

If multiple elements are valid for a file name used in your program, EGL uses the first element that applies. A series of association elements, for example, might be characterized by the following values for file name, in order:

```

myFile
myFile*
*
```

Consider the element associated with the last value, where the value of `myFile` is only an asterisk. Such an element could apply to any file; but in relation to a particular file, the last element applies only if the previous elements do not. If your program references `myFile01`, for instance, the linkage specified in the second element supersedes the third element to define how the reference is handled.

At generation time, EGL selects a particular association element, along with the first entry that is appropriate. An entry is appropriate in either of two cases:

- A match exists between the target system for which you are generating, on the one hand, and the **system** property, on the other; or
- The **system** property has the following value:  
any

If you are generating for AIX, for example, EGL uses the first entry that refers to **aix** or to **any**.

**File types:** A file type determines what properties are necessary for a given entry in an association element. The next table describes the EGL file types.

File type	Description
ibmcobol	A VSAM file accessed remotely by an EGL-generated Java program. For details on specifying the system name in this case, see VSAM support.
mq	An MQSeries message queue; for details on how to work with such a queue, see MQSeries support.
seqws	A serial file accessed by an EGL-generated Java program.
spool	A spool file on AIX or Linux.

**Record types and VSAM:** Each of three types of fixed records is appropriate for accessing a VSAM data set, but only if the file type in the association element for the record is `ibmcobol`, `vsam`, or `vsamrs`:

- If the fixed record is of type `indexedRecord`, the VSAM data set is a Key Sequenced Data Set with a primary or alternate index
- If the fixed record is of type `relativeRecord`, the VSAM data set is a Relative Record Data Set
- If the fixed record is of type `serialRecord`, the VSAM data set is an Entry Sequenced Data Set

**For further details:** For further details on resource associations, see these topics:

- *Record and file type cross-reference*
- *Association elements*

### Related concepts

“Fixed record parts” on page 136  
 “MQSeries support” on page 336  
 “Parts” on page 19  
 “Record types and properties” on page 138  
 “Record parts” on page 135  
 “VSAM support” on page 335

### Related task

“Adding a resource associations part to an EGL build file” on page 397  
 “Editing a resource associations part in an EGL build file” on page 397  
 “Removing a resource associations part from an EGL build file” on page 398

### Related reference

“Association elements” on page 457  
 “Record and file type cross-reference” on page 860  
 “recordName.resourceAssociation” on page 985  
 “resourceAssociations” on page 486  
 “system” on page 494  
 “printerAssociation” on page 1059

## Logical unit of work

When you change resources that are categorized as *non-recoverable* (such as serial files on Windows 2000), your work is relatively permanent; neither your code nor EGL runtime services can simply rescind the changes. When you change resources that are categorized as *recoverable* (such as relational databases), your code or EGL runtime services either can commit the changes to make the work permanent or can rollback the changes to return to content that was in effect when changes were last committed.

Recoverable resources are as follows:

- Relational databases
- CICS queues and files that are configured to be recoverable
- MQSeries message queues, unless your MQSeries record specifies otherwise, as described in *MQSeries support*

A *logical unit of work* identifies input operations that are either committed or rolled back as a group. A unit of work begins when your code changes a recoverable resource; and ends when the first of these events occurs:

- Your code invokes the system function **sysLib.commit** or **sysLib.rollback** to commit or roll back the changes
- EGL runtime services performs a rollback in response to a hard error that is not handled in your code; in this case, all the programs in the run unit are removed from memory
- An implicit commit occurs, as happens in the following cases--
  - A program issues a **show** statement.
  - The top-level program in a run unit ends successfully, as described in *Run unit*.
  - A Web page is displayed, as when a PageHandler issues a **forward** statement.
  - A program issues a **converse** statement and any of the following applies:
    - You are not in VisualAge Generator compatibility mode, and the program is a segmented program
    - **ConverseVar.commitOnConverse** is set to 1
    - You are in VisualAge Generator compatibility mode, and **ConverseVar.segmentedMode** is set to 1

**Unit of work for Java:** In a Java run unit, the details are as follows:

- When any of the Java programs ends with a hard error, the effect is equivalent to performing rollbacks, closing cursors, and releasing locks.
- When the run unit ends successfully, EGL performs a commit, closes cursors, and releases locks.
- You can use multiple connections to read from multiple databases, but you should update only one database in a unit of work because only a one-phase commit is available. For related information, see *VGLib.connectionService*.
- When an EGL-generated program is accessed by way of an EGL-generated EJB session bean, transaction control may be affected by a transaction attribute (also called the container transaction type), which is in the deployment descriptor of the EJB session bean. The transaction attribute affects transaction control only when the linkage options part, callLink element, property **remoteComType** for the call is direct, as described in *remoteComType in callLink element*.

The EJB session bean is generated with transaction attribute REQUIRED, but you can change the value at deployment time. For details on the implications of the transaction attribute, see your Java documentation.

#### Related concepts

“MQSeries support” on page 336

“Run unit” on page 866

“SQL support” on page 277

#### Related tasks

“Setting up a J2EE JDBC connection” on page 445

“Understanding how a standard JDBC connection is made” on page 309



### Related reference

"Default database" on page 298  
"commit()" on page 1024  
"connectionService()" on page 1047  
"rollback()" on page 1036  
"Java wrapper classes" on page 652  
"luwControl in callLink element" on page 506  
"remoteComType in callLink element" on page 511  
"sqlDB" on page 490

## Adding a resource associations part to an EGL build file

A resource associations part relates a file name with a system resource name on the target platform where you intend to deploy the generated code. You can add a resource associations part to an EGL build file. For details, see *Resource associations and file types*. To add a resource associations part, do the following:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
3. In the Outline view, right-click on the build file, then click **Add Part**.
4. Click **Resource Associations**, then click **Next**.
5. Choose a name for your resource associations part that conforms to EGL part name conventions. In the Name field, type the name of your resource associations part.
6. In the Description field, type a description of your part.
7. Click **Finish**. The resource associations part is added to the EGL build file and the resource associations part page is opened in the EGL build parts editor.

### Related concepts

"Build descriptor part" on page 383  
"Resource associations and file types" on page 393

### Related tasks

"Editing a resource associations part in an EGL build file"  
"Removing a resource associations part from an EGL build file" on page 398

### Related reference

"EGL build-file format" on page 462  
"Naming conventions" on page 778

## Editing a resource associations part in an EGL build file

A resource associations part relates a file name with a system resource name on the target platform where you intend to deploy the generated code.

To edit a resource associations part, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.



3. In the Outline view, right-click a resource associations part and click **Open**. The editor displays the current part definition.
4. To add a new Association element to the part, click **Add Association** or press the Insert key, and type the logical file name or select a logical file name.
5. To change the default system name associated with your logical file name, you can either:
  - Select the corresponding row in the Association elements list, then click the name once to put the field into edit mode. Select the new system name from the System drop-down list.
  - In the Properties of selected system entries list, click the system property once to put the Value field associated with that property into edit mode. Select the new system name from the Value drop-down list.
6. To change the default file type associated with your logical file name, you can either:
  - Select the row in the Association elements list that corresponds to your logical file name, then click the name once to put the field into edit mode. Select the new file type from the File Type drop-down list.
  - Select the row in the Association elements list that corresponds to your logical file name. In the Properties of selected system entries list, click the fileType property once to put the Value field associated with that property into edit mode. Select the file type from the Value drop-down list.
7. Modify the resource associations as needed.
  - To associate more than one system and set of related properties with a logical file name, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Add System**. The added row is now selected and available for editing.
  - To remove a system and related properties from an associated logical file name, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Remove** or press the Delete key.
  - To remove a logical file name and any associated systems, select the row in the Association elements list that corresponds to your logical file name. At the bottom of the Association elements list, click **Remove** or press the Delete key.

#### Related concepts

"Build descriptor part" on page 383

"Resource associations and file types" on page 393

#### Related tasks

"Adding a resource associations part to an EGL build file" on page 397

"Removing a resource associations part from an EGL build file"

#### Related reference

"EGL build-file format" on page 462

### Removing a resource associations part from an EGL build file

To remove a resource associations part from an EGL build file, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**

2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu
3. In the Outline view, right-click on the resource associations part, then click **Remove**

#### Related concepts

“Resource associations and file types” on page 393

#### Related tasks

“Adding a resource associations part to an EGL build file” on page 397

“Editing a resource associations part in an EGL build file” on page 397

## Setting up call and transfer options

### Linkage options part

A *linkage options* part specifies details on the following issues:

- How a generated program or wrapper calls other generated code
- How a generated program transfers asynchronously to another generated program

**Specifying when linkage options are final:** You can choose between two alternatives:

- The linkage options specified at generation time are in effect at run time; or
- The linkage options specified in a linkage properties file at deployment time are in effect at run time. Although you can write that file by hand, EGL generates it in this situation:
  - You set the linkage options property **remoteBind** to **RUNTIME**; and
  - You generate a Java program or wrapper with the build descriptor option **genProperties** set to **GLOBAL** or **PROGRAM**.

For details on using the file, see Deploying a linkage properties file. For details on customizing the file, see Linkage properties file (reference).

**Elements of a linkage options part:** The linkage options part is composed of a set of elements, each of which has a set of properties and values. The following types of elements are available:

- A **callLink** element specifies the linkage conventions that EGL uses for a given call.

The **callLink** element always applies to a called program. The following relationships are in effect:

- If the **callLink** element refers to the program that you are generating, that element helps determine whether to generate a Java wrapper that allows access to the program from native Java code; for an overview, see *Java wrapper*. If you indicate that the Java wrapper accesses the program by way of an EJB session bean, the **callLink** element also causes generation of an EJB session bean.
- If you are generating a Java program and if the **callLink** element refers to a program that is called by that program, the **callLink** element specifies how the call is implemented; for example, whether the call is local or remote. If you indicate that the calling Java program makes the call through an EJB session bean, the **callLink** element causes generation of an EJB session bean.
- An *asynchLink* element specifies how a generated program transfers asynchronously to another program, as occurs when the transferring program invokes the system function `sysLib.startTransaction`.

- A *transferToProgram* element specifies how a generated COBOL program transfers control to a program and ends processing. This element is not used for Java output and is meaningful only for a main program that issues a **transfer** statement of the type *transfer to program*.
- A *transferToTransaction* element specifies how a generated program transfers control to a transaction and ends processing. This element is meaningful only for a main program that issues a **transfer** statement of the type *transfer to transaction*. The element is unnecessary, however, when the target program is generated with VisualAge Generator or (in the absence of an alias) with EGL.

**Identifying the programs or records to which an element refers:** In each element, a property (for example, **pgmName**) identifies the programs or records to which the element refers; and unless otherwise stated, the value of that property can be a valid name, an asterisk, or the beginning of a valid name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names.

Consider a *callLink* element that includes the following value for the **pgmName** property:

```
myProg*
```

That element pertains to any EGL program part that begins with the letters *myProg*.

If multiple elements are valid, EGL uses the first element that applies. A series of *callLink* elements, for example, might be characterized by these **pgmName** values, in order:

```
YourProgram
YourProg*
*
```

Consider the element associated with the last value, where the value of **pgmName** is only an asterisk. Such an element could apply to any program; but in relation to a particular program, the last element applies only if the previous elements do not. If your program calls *YourProgram01*, for instance, the linkage specified in the second element (*YourProg\**) supersedes the third element (\*) to define how EGL handles the call.

In most cases, elements with more specific names should precede those with more general names. In the previous example, the element with the asterisk is appropriately positioned to provide the default linkage specifications.

#### Related concepts

"Java wrapper" on page 390

"Parts" on page 19

#### Related tasks

"Adding a linkage options part to an EGL build file" on page 401

"Deploying a linkage properties file" on page 447

"Editing the *asynchLink* element of a linkage options part" on page 403

"Editing the *callLink* element of a linkage options part" on page 401

"Editing the transfer-related elements of a linkage options part" on page 404

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

### Related reference

"asynchLink element" on page 459

"call" on page 665

"callLink element" on page 499

"linkage" on page 484

"Linkage properties file (details)" on page 764

"startTransaction()" on page 1041

"transfer" on page 752

"transferToTransaction element" on page 1088

## Adding a linkage options part to an EGL build file

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To add this type of part, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
3. In the Outline view, right-click on the build file, then click **Add Part**.
4. Click **Linkage Options**, then click **Next**.
5. Choose a name for your linkage options part that adheres to EGL part name conventions. In the Name field, type the name of your linkage options part.
6. In the Description field, type a description of your part.
7. Click **Finish**. The linkage options part is added to the EGL file and the linkage options part page is opened in the EGL build parts editor.

### Related concepts

"Linkage options part" on page 399

### Related tasks

"Editing the asynchLink element of a linkage options part" on page 403

"Editing the callLink element of a linkage options part"

"Editing the transfer-related elements of a linkage options part" on page 404

"Removing a linkage options part from an EGL build file" on page 405

### Related reference

"EGL build-file format" on page 462

"Naming conventions" on page 778

## Editing the callLink element of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To edit the part's callLink element, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**

2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
3. In the Outline view, right-click a linkage options part and click **Open**. The EGL part editor displays the current part declaration.
4. Click the Show CallLink Elements button on the editor toolbar.
5. To add a new CallLink element, click **Add** or press the Insert key, and type the Program Name (pgmName) or select a program name from the Program Name drop-down list.
6. To change the default call type associated with your program name, you can either:
  - Select the corresponding row in the CallLink elements list, then click the Type field (localCall, remoteCall, ejbCall) once to put the field into edit mode. Select the new call type from the Type drop-down list.
  - In the Properties of selected callLink elements list, click the type property once to put the Value field associated with that property into edit mode. Select the new call type from the Value drop-down list.
7. Other properties associated with your program name are listed in the Properties of selected callLink elements list based on the call type. To change the value of one of these properties, select the program name. In the Properties of selected callLink elements list, click the property you want to define once to put the Value field associated with that property into edit mode. Define the new value by selecting an option in the Value drop-down list, or by typing the new value in the Value field. For some properties, you can only select an option in a drop-down list. For other properties, you can only type a value in the Value field.
8. Modify the CallLink elements list as needed:
  - To reposition a callLink element, select an element and click either **Move Up** or **Move Down**.
  - To remove a callLink element, select the element and click **Remove** or press the Delete key.

#### Related concepts

"Linkage options part" on page 399

#### Related tasks

"Adding a linkage options part to an EGL build file" on page 401

"Editing the asynchLink element of a linkage options part" on page 403

"Editing the transfer-related elements of a linkage options part" on page 404

"Removing a linkage options part from an EGL build file" on page 405

#### Related reference

"asynchLink element" on page 459

"callLink element" on page 499

"EGL build-file format" on page 462

"Linkage properties file (details)" on page 764

**Enterprise JavaBean (EJB) session bean:** An EJB session bean comprises the following components:

- Home interface, which gives a client access to the EJB session bean at run time
- Remote bean interface, which lists the methods that are directly available to that client

- Bean implementation, which contains the logic that is indirectly available to that client

An EJB session bean is an intermediary between one program and another or between an EGL Java wrapper and a program. Generation of the EJB session bean largely depends on settings in the linkage options part that is used at generation time. For details, see *Linkage options part*; in particular, the overview of the **callLink** element.

For details on the output file names, see *Generated output (reference)*.

#### Related concepts

“Generated output” on page 625

“Linkage options part” on page 399

#### Related reference

“Generated output (reference)” on page 626

### Editing the **asynchLink** element of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how it accesses files. To edit the part’s **asynchLink** element, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
3. In the Outline view, right-click a linkage options part and click **Open**. The EGL build parts editor displays the current part declaration.
4. Click the Show AsynchLink Elements button on the editor toolbar.
5. To add a new AsynchLink element, click **Add** or press the Insert key, and type the Record Name (recordName) or select a record name from the Record Name drop-down list.
6. To change the default linkage type associated with your record name, you can either:
  - Select the corresponding row in the AsynchLink Elements list, then click the Type field (localAsynch, remoteAsynch) once to put the field into edit mode. Select the new linkage type from the Type drop-down list.
  - In the Properties of selected asynchLink elements list, click the type property once to put the Value field associated with that property into edit mode. Select the new linkage type from the Value drop-down list.
7. Other properties associated with your record name are listed in the Properties of selected asynchLink elements list based on the linkage type. To change the value of one of these properties, select the record name. In the Properties of selected asynchLink elements list, click the property you want to define once to put the Value field associated with that property into edit mode. Define the new value by selecting an option in the Value drop-down list, or by typing the new value in the Value field. For some properties, you can only select an option in a drop-down list. For other properties, you can only type a value in the Value field.
8. Modify the asynchLink elements list as needed:

- To reposition an asynchLink element, select an element and click either **Move Up** or **Move Down**.
- To remove an asynchLink element, select an element and click **Remove** or press the Delete key.

#### Related concepts

“Linkage options part” on page 399

#### Related tasks

“Adding a linkage options part to an EGL build file” on page 401

“Editing the callLink element of a linkage options part” on page 401

“Editing the transfer-related elements of a linkage options part”

“Removing a linkage options part from an EGL build file” on page 405

#### Related reference

“asynchLink element” on page 459

“EGL build-file format” on page 462

“Linkage properties file (details)” on page 764

“startTransaction()” on page 1041

### Editing the transfer-related elements of a linkage options part

A linkage options part describes how a generated EGL program implements calls and transfers and how the program accesses files. To edit the part’s transfer-related elements, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu.
3. In the Outline view, right-click a linkage options part and click **Open**. The EGL build parts editor displays the current part declaration.
4. Click the Show TransferLink Elements button on the editor toolbar. The Transfer to Program and Transfer to Transaction lists display.
5. To edit the Transfer to Program list, do as follows:
  - a. At the bottom of the Transfer to Program list, click **Add** or press the Insert key, and type the From Program (fromPgm) name or select a program name from the From Program name drop-down list.
  - b. To edit the To Program (toPgm) name, select the corresponding row in the Transfer to Program list, then click the To Program field once to put the field into edit mode. Type the program name or select a program name from the To Program drop-down list.
  - c. If an alias name is needed, select the corresponding row in the Transfer to Program list, then click the Alias field once to put the field into edit mode. Type the alias name.
  - d. To change the default linkage type associated with your program name, select the corresponding row in the Transfer to Program list, then click the Link Type (linkType) field once to put the field into edit mode. Select the new linkage type from the Link Type drop-down list.
6. To edit the Transfer to Transaction list, do as follows:



- a. At the bottom of the Transfer to Transaction list, click **Add** or press the Insert key, and type the To Program (toPgm) name or select a program name from the To Program name drop-down list.
- b. If an alias name is needed, select the corresponding row in the Transfer to Transaction list, then click the Alias field once to put the field into edit mode. Type the alias name.
- c. To edit the Externally Defined property associated with your program name, select the corresponding row in the Transfer to Transaction list, then click the Externally Defined field once to put the field into edit mode. Select the externally defined property from the Externally Defined property drop-down list.
- d. Modify the Transfer to Transaction list as needed:
  - To reposition a transferToTransaction element, select an element and click either **Move Up** or **Move Down**.
  - To remove a transferToTransaction element, select an element and click **Remove** or press the Delete key.

**Note:** Transfer to Program is only relevant in these products:

- Rational Application Developer for iSeries
- Rational Application Developer for z/OS

#### **Related concepts**

"Linkage options part" on page 399

#### **Related tasks**

"Adding a linkage options part to an EGL build file" on page 401

"Editing the asynchLink element of a linkage options part" on page 403

"Editing the callLink element of a linkage options part" on page 401

"Removing a linkage options part from an EGL build file"

#### **Related reference**

"EGL build-file format" on page 462

"transferToTransaction element" on page 1088

### **Removing a linkage options part from an EGL build file**

To remove a linkage options part from an EGL build file, do as follows:

1. To open an EGL build file with the EGL build parts editor, do as follows in the Project Explorer:
  - a. Right-click on the EGL build file
  - b. Select **Open With > EGL Build Parts Editor**
2. If the Outline view is not displayed, open that view by selecting **Show View > Outline** from the Window menu
3. In the Outline view, right-click on the linkage options part, then click **Remove**

#### **Related concepts**

"Linkage options part" on page 399

#### **Related tasks**

"Adding a linkage options part to an EGL build file" on page 401

"Editing the asynchLink element of a linkage options part" on page 403

"Editing the callLink element of a linkage options part" on page 401



“Editing the transfer-related elements of a linkage options part” on page 404

## Setting up references to other EGL build files

### Adding an import statement to an EGL build file

Import statements allow EGL build files to reference parts in other build files. See *Import* for more information on the import feature.

To add an import statement to an EGL build file, do as follows:

1. Open an EGL build file with the EGL build parts editor. If you do not have a file open, do this in the Project Explorer:
  - a. Right-click on the build file in the Project Explorer
  - b. Select **Open With > EGL Build Parts Editor**
2. Click the **Imports** tab in the build parts editor.
3. Click the **Add** button.
4. Type or select the name of the file or folder to import, then click **OK**.

#### Related concepts

“Import” on page 33

#### Related tasks

“Editing an import statement in an EGL build file”

“Removing an import statement from an EGL build file”

### Editing an import statement in an EGL build file

To edit an import statement in an EGL build file, do as follows:

1. Open the EGL build file with the EGL build parts editor. If you do not have a file open, do this in the Project Explorer:
  - a. Right-click on the build file in the Project Explorer
  - b. Select **Open With > EGL Build Parts Editor**
2. Click the **Imports** tab in the build parts editor. The import statements are displayed.
3. Select the import statement you want to change, then click the **Edit** button.
4. Type or select the name of the file or folder to import, then click **OK**.

#### Related concepts

“Import” on page 33

#### Related tasks

“Adding an import statement to an EGL build file”

“Removing an import statement from an EGL build file”

### Removing an import statement from an EGL build file

To remove an import statement in an EGL build file, do as follows:

1. Open the EGL build file with the EGL build parts editor. If you do not have a file open, do this in the Project Explorer:
  - a. Right-click on the build file in the Project Explorer
  - b. Select **Open With > EGL Build Parts Editor**
2. Click the **Imports** tab in the build parts editor. The import statements are displayed.

3. Select the import statement you want to remove, then click the **Remove** button.

#### **Related concepts**

"Import" on page 33

#### **Related tasks**

"Adding an import statement to an EGL build file" on page 406

"Editing an import statement in an EGL build file" on page 406

---

## **Editing an EGL build path**

For overview material, see these topics:

- *References to parts*
- *EGL build path and eglpath*

To include projects in the EGL project path, follow these steps:

1. In the Project Explorer, right-click on a project that you want to link to other projects, then click **Properties**.
2. Select the **EGL Build Path** properties page.
3. A list of all other projects in your workspace is displayed in the **Projects** tab. Click the check box beside each project you want to reference.
4. To put the projects in a different order or to export any of them, click the **Order and Export** tab and do as follows--
  - To change the position of a project in the build-path order, select the project and click the **Up** and **Down** buttons.
  - To export a project, select the related check box. To handle all the projects at once, click the **Select All** or **Deselect All** button.
5. Click **OK**.

#### **Related concepts**

"EGL projects, packages, and files" on page 15

"References to parts" on page 23

"Import" on page 33

"Parts" on page 19

#### **Related reference**

"EGL build path and eglpath" on page 571

"References to parts" on page 23

"Import" on page 33

"Parts" on page 19



---

# Generating, preparing, and running EGL output

---

## Generation

Generation is the creation of output from EGL parts.

You can generate output in the Workbench, from the Workbench batch interface, or from the EGL Software Development Kit (EGL SDK). The EGL SDK provides a batch interface for file-based generation that is independent of the Workbench.

Generation uses the saved versions of your EGL files.

### Related concepts

“Development process” on page 9

“Generated output” on page 625

### Related tasks

“Generating Java wrappers” on page 390

### Related reference

“Build descriptor options” on page 464

“Generated output (reference)” on page 626

## Generation of Java code into a project

If you are generating a Java program or wrapper, it is recommended (and in some cases required) that you set build descriptor option **genProject**, which causes generation into a project.

EGL provides various services for you when you generate into a project. The services vary by project type, as do your next tasks:

### Application client project

When you generate into an application client project, EGL does as follows:

- Provides preparation-time access to EGL jar files (fda6.jar and fdaj6.jar) by adding the following entries to the project's Java build path:

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

For details on the variable at the beginning of each entry, see *Setting the variable EGL\_GENERATORS\_PLUGINDIR*.

- Provides runtime access to the EGL jar files:
  - Imports the jar files into each enterprise application project that references the application client project
  - Updates the manifest in the application client project so that the jar files in an enterprise application project are available
- Puts runtime values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*

Your next tasks are as follows:

1. If you are calling the generated program by way of TCP/IP, provide runtime access to a listener, as described in *Setting up the TCP/IP listener*

2. Provide access to non-EGL jar files
3. Now that you have placed output files in a project, continue setting up the J2EE runtime environment

### EJB project

When you generate into an EJB project, EGL does as follows:

- Provides preparation-time access to EGL jar files (fda6.jar and fdaj6.jar) by adding the following entries to the project's Java build path:

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

For details on the environment variable at the beginning of each entry, see *Setting the variable EGL\_GENERATORS\_PLUGINDIR*.

- Provides runtime access to the EGL jar files:
  - Imports fda6.jar and fdaj6.jar into each enterprise application project that references the EJB project
  - Updates the manifest in the EJB project so that fda6.jar and fdaj6.jar in an enterprise application project are available at run time
- Assigns the JNDI name automatically so that the EGL runtime code can access the EJB code; but this step occurs only when you generate an EJB session bean.
- In most cases, puts runtime values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*.

EGL does not put runtime values into the deployment descriptor if EGL cannot find the necessary session element in the deployment descriptor. This situation occurs, for example, when the Java program is generated before the wrapper or when the build descriptor option **sessionBeanID** is set to a value that is not found in the deployment descriptor. For details on session elements, see *sessionBeanID*.

Your next tasks are as follows:

1. Provide access to non-EGL jar files
2. Generate deployment code
3. Now that you have placed output files in a project, continue setting up the J2EE runtime environment

### J2EE Web project

EGL does as follows:

- Provides access to EGL jar files by importing fda6.jar and fdaj6.jar into the project's Web Content/WEB-INF/lib folder
- Puts runtime values into the deployment descriptor so that you can avoid cutting and pasting entries from a generated J2EE environment file; for an overview of this subject, see *Setting deployment-descriptor values*

Your next tasks are as follows:

1. Providing access to non-EGL jar files
2. Now that you have placed output files in a project, continue as described in *Setting up the J2EE runtime environment for EGL-generated code*

### Java project

If you are generating into a non-J2EE Java project for debugging or production purposes, EGL does as follows:

- Provides access to EGL jar files (fda6.jar and fdaj6.jar) by adding the following entries to the project's Java build path:

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

For details on the variable at the beginning of each entry, see *Setting the variable EGL\_GENERATORS\_PLUGINDIR*.

- Generates a properties file, but only if the build descriptor includes the following option values:
  - **genProperties** is set to GLOBAL or PROGRAM; and
  - **J2EE** is set to NO.

If you request a global properties file (**rununit.properties**), EGL places that file in the Java source folder, which is the folder that contains the Java packages. (The Java source folder may be either a folder within the project or the project itself.) If you request a program properties file instead, EGL places that file in the folder that contains the program.

At run time, values in the program properties file are used to set up a standard JDBC connection. For details, see *Understanding how a standard JDBC connection is made*.

Now that you have placed output files in a project, do as follows:

- If your program accesses a relational database, make sure that your Java build path includes the directory where the driver is installed. For DB2, for example, specify the directory that contains db2java.zip.
- If your code accesses MQSeries, provide access to non-EGL jar files
- Place a linkage properties file in the module

For details on the consequence of generating into a non-existent project, see *genProject*.

#### Related tasks

"Generating deployment code for EJB projects" on page 423

"Deploying a linkage properties file" on page 447

"Setting deployment-descriptor values" on page 438

"Providing access to non-EGL jar files" on page 448

"Setting the variable EGL\_GENERATORS\_PLUGINDIR" on page 423

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

"Understanding how a standard JDBC connection is made" on page 309

#### Related reference

"genProject" on page 478

"sessionBeanID" on page 488

## Build

When you are working in an EGL or EGL Web project, the word *build* does not (in general) refer to code generation.

The following menu options have a distinct meaning:

#### Build project

Builds a subset of the project--

1. Validates all EGL files that have changed in the project since the last build

2. Generates PageHandlers that were changed since the prior PageHandler generation
3. Compiles any Java source that changed since the last compile

The menu option **Build Project** is available only if you have not set the Workbench preference **Perform build automatically on resource modification**. If you *have* set that preference, the actions described earlier occur whenever you save an EGL file.

#### **Build all**

Conducts the same actions as **Build project**, but for every open project in the workspace.

#### **Rebuild project**

Acts as follows--

1. Validates all the EGL files in the project
2. Generates all PageHandlers in the project
3. Compiles any Java source that changed since the last compile

#### **Rebuild all**

Conducts the same actions as **Rebuild project**, but for every open project in the workspace.

When you generate code into a project, a Java compile occurs locally in the following situations:

- When you build or rebuild the project; or
- When you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**.

When you generate code into a directory, EGL optionally creates a *build plan*, which is an XML file that includes the following details:

- The location of any files that will be transferred to another machine;
- Other information needed for the transfer, which occurs by way of TCP/IP; and
- A Java compile statement.

Preparation of generated output on a remote platform requires that a build server be running on that platform.

You may wish to create a build plan and to invoke that plan at a later time. For details, see *Invoking a build plan after generation*.

#### **Related concepts**

“Build descriptor part” on page 383

“Build plan” on page 413

“Build server” on page 427

“Development process” on page 9

#### **Related tasks**

“Creating a build file” on page 383

“Invoking a build plan after generation” on page 419

#### **Related reference**

“Build descriptor options” on page 464

## Building EGL output

To build EGL output for Java programs or wrappers, complete the following steps:

1. Generate Java source code into a project or directory:
  - If you generate into a project (as is recommended) and if your Eclipse preferences are set to build automatically on resource modification, the workbench prepares the output.
  - If you generate into a directory, the generator's distributed build function prepares the output.
2. Prepare the generated output. This step is done automatically unless you set build descriptor option **buildPlan** or **prep** to no.

### Related concepts

"Build" on page 411

"Development process" on page 9

"EGL projects, packages, and files" on page 15

"Generation of Java code into a project" on page 409

"Generated output" on page 625

"Generation" on page 409

### Related tasks

"Creating an EGL source file" on page 130

"Processing Java code that is generated into a directory" on page 420

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

### Related reference

"Generated output (reference)" on page 626

## Build plan

The build plan is an XML file that makes the following details available at preparation time:

- What files need to be processed on the build machine
- What build scripts are needed to process them
- Where outputs are to be placed

The build plan resides on the development platform and informs the build client of all the build steps. For each step a request is made of the build server.

EGL produces a build plan whenever you generate a Java program or wrapper, unless you set the build descriptor option **buildPlan** to NO.

For details on the name of the build plan, see *Generated output (reference)*.

### Related concepts

"Build script" on page 426

"Generated output" on page 625

### Related reference

"buildPlan" on page 469

"Generated output (reference)" on page 626



## Java program, PageHandler, and library

When you request that a program part be generated as a Java program, or when you request that a pageHandler or Java-related library part be generated, EGL produces a class and a file for each of the following:

- The program, pageHandler, or library part
- Each record declared either in that part itself or in any function that is invoked directly or indirectly by that part
- Each data table, form group, and form that is used

For details on the class names, see *Output of Java program generation*.

### Related tasks

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

### Related reference

“Generated output (reference)” on page 626

“Output of Java program generation” on page 781

## Results file

The results file contains status information on the code-preparation steps that were done on the target environment. You receive the file only if EGL attempts to prepare generated output.

Preparation occurs automatically when you generate into a directory and use the following build descriptor options:

- **prep** is set to YES
- **buildPlan** is set to YES

For details on the name of the results file, see *Generated output (reference)*.

### Related concepts

“Build descriptor part” on page 383

“Generated output” on page 625

### Related tasks

“Processing Java code that is generated into a directory” on page 420

### Related reference

“Generated output (reference)” on page 626

“buildPlan” on page 469

“prep” on page 485

---

## Generating in the workbench

Generating in the workbench is accomplished with either the generation wizard or the generation menu item. When you select the generation menu item, EGL uses the default build descriptor. If you have not selected a default build descriptor, use the generation wizard. For details on selecting the default build descriptor, see *Setting the default build descriptors*.

To generate in the workbench by invoking the generation wizard, do as follows:

1. Right-click on a resource name (project, folder, or file) in the Project Explorer.
2. Select the **Generate With Wizard...** option.

The generation wizard includes four pages:

1. The first page displays a list of parts to generate based on the selection that the user made to start the generation process. You must select at least one part from the list before you can continue to the next page. The interface provides buttons to let you select or deselect all of the parts in the list.
2. The second page lets you choose a build descriptor or build descriptors to be used to generate the parts selected on the first page. You have two options:
  - Choose from a drop-down list of all the build descriptors that are in the workspace, and use that build descriptor to generate all of the parts.
  - Select a build descriptor for each of the parts selected on the first page. You use a table to select the build descriptors for each part. The first column of the table displays the part names; the second column displays a drop-down list of build descriptors for each part.
3. The third page lets you set user IDs and passwords for both the destination machine and the SQL database used in the generation process, if IDs and passwords are necessary. These user IDs and passwords, if any, override the ones listed in the specified build descriptor for each part being generated. You may want to set user IDs and passwords on this page rather than the build descriptor to avoid keeping sensitive information in persistent storage.
4. The fourth page lets you create a command file that you can use for generating an EGL program outside of the workbench. You can reference the command file in the workbench batch interface (by using the command EGLCMD) or in the EGL SDK (by using the command EGLSDK).

To create a command file, do as follows:

- a. Select the Create a Command File checkbox
  - b. Specify the name of the output file, either by typing the fully qualified path or by clicking **Browse** and using the standard Windows procedure for selecting a file
  - c. Select or clear the Automatically Insert EGL Path (eglpth) check box to specify whether you want to include the EGL project path in the command file, as the initial value for eglpth; for details, see *EGL command file*
  - d. Select a radio button to indicate whether to avoid generating output when creating the command file
5. Click **Finish**.

To generate in the workbench using the generation menu item, do one of the following sets of steps:

1. Select one or more resource names (project, folder, or file) in the Project Explorer. To select multiple resource names, hold down the **Ctrl** key as you click.
2. Right-click, then select the **Generate** menu option.

or

1. Double-click on a resource name (project, folder, or file) in the Project Explorer. The file opens in the EGL editor.
2. Right-click inside the editor pane, then select **Generate**.

#### Related concepts

“Generation in the workbench” on page 416

#### Related tasks

“Setting the default build descriptors” on page 118

#### Related reference

"EGLCMD" on page 572

"EGLSDK" on page 583

"Generated output (reference)" on page 626

## Generation in the workbench

To generate output in the Workbench, do the following:

- Load parts to generate, along with any parts that are referenced during generation.
- Select parts to generate. If you invoke the generation process for a file, folder, package, or project, EGL creates output for every generatable part that is in the container you selected.
- Initiate generation.
- Monitor progress.

To make generation easier, it is recommended that you first select a default build descriptor from these types:

- Debug build descriptor (as appropriate when you are using the EGL debugger)
- Target system build descriptor (as used for generating parts and deploying them in a runtime environment)

For details on how to select a default build descriptor, see *Setting the default build descriptors*.

You can identify each of two build descriptors (debug and target system) in these ways:

- As a property at the file, folder, package, and project level
- As a Workbench preference

A lower-level build descriptor of a particular kind takes precedence over any higher-level build descriptor of the same kind. For example, a target system build descriptor that was assigned to the current package takes precedence over a target system build descriptor that was assigned to the project. However, a master build descriptor takes precedence over all others, as described in *Build descriptor part*.

When you generate a VGWebTransaction program and want to generate a VGUIRecord with a different set of options, set the build descriptor option **secondaryTargetBuildDescriptor**.

For details on initiating generation, see *Generating in the workbench*.

#### Related concepts

"Build descriptor part" on page 383

"Development process" on page 9

"Generated output" on page 625

"Parts" on page 19

#### Related tasks

"Generating in the workbench" on page 414

"Setting the default build descriptors" on page 118

#### Related reference

"Generated output (reference)" on page 626

---

## Generating from the workbench batch interface

To generate from the Workbench batch interface, do as follows:

1. Make sure that your Java classpath provides access to these jar files--
  - startup.jar, which is in the following directory:  
*installationDir*\eclipse  
*installationDir*  
The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.
  - eglutil.jar, which is in the following directory:  
*installationDir*\egl\eclipse\plugins\com.ibm.etools.egl.utilities\_*version*\runtime  
*installationDir*  
The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.
- version*  
The installed version of the plugin; for example, 6.0.0
2. Make sure that a workspace contains the projects and EGL parts that are required for generation.
3. Develop an EGL command file.
4. Invoke the command EGLCMD, possibly in a larger batch job that generates, runs, and tests the code. You specify the workspace of interest when invoking EGLCMD.

### Related concepts

“Generation from the workbench batch interface”

### Related reference

“EGLCMD” on page 572

“EGL command file” on page 575

## Generation from the workbench batch interface

The workbench batch interface is a feature that lets you generate EGL output from a batch environment that can access the workbench. The workbench does not need to be running. The generation of EGL code can access only projects and EGL parts that were previously loaded into a workspace.

To invoke the interface, use the batch command EGLCMD, which references both a workspace and an EGL command file.

### Related concepts

“Development process” on page 9

“Generated output” on page 625

### Related tasks

“Generating from the workbench batch interface”

**Related reference**  
“EGLCMD” on page 572

---

## Generating from the EGL Software Development Kit (SDK)

To generate from the EGL SDK, do as follows:

1. Make sure that Java 1.3.1 (or a higher level) is on the machine where you will generate code. An appropriate level of Java code is installed automatically on the machine where you install EGL. The Java levels on the generation and target machines must be compatible.
2. Make sure that `eglbatchgen.jar` is in your Java classpath. The jar file is in the following directory:

`installationDir\bin`

`installationDir`

The product installation directory, such as `C:\Program Files\IBM\RSPD\6.0`. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

3. Make sure that the EGL SDK can access the EGL files that are required for generation
4. Optionally, develop an EGL command file
5. Invoke the command `EGLSDK`, possibly in a larger batch job that generates, runs, and tests the code

**Related concepts**  
“Generation from the EGL Software Development Kit (SDK)”  
“EGL projects, packages, and files” on page 15

**Related reference**  
“EGL build path and `eglp`” on page 571  
“EGLCMD” on page 572  
“EGL command file” on page 575  
“EGLSDK” on page 583

## Generation from the EGL Software Development Kit (SDK)

The EGL software development kit (SDK) is a feature that lets you generate output in a batch environment, even when you lack access to the following aspects of the Rational Developer product:

- The graphical user interface
- The details on how projects are organized

You can use the EGL SDK to trigger generation from a software configuration management (SCM) tool such as Rational ClearCase®, perhaps as part of a batch job that is run after normal working hours.

To invoke the EGL SDK, you use the command `EGLSDK` in a batch file or at a command prompt. The command invocation itself can take either of two forms:

- It can specify an EGL file and build descriptor. In this case, if you want to cause multiple generations you write multiple commands.
- Alternatively, the invocation can reference an EGL command file that includes the information necessary to cause one or more generations.

However you organize your work, you can specify a value for *eglp*path, which is a list of directories that are searched when the EGL SDK uses an import statement to resolve a part reference. Also, you must specify the build descriptor option **genDirectory** instead of **genProject**.

The prerequisites and process for using EGLSDK are described in *Generating from the EGL SDK*. For details on the command invocation, see *EGLSDK*.

#### Related concepts

"Development process" on page 9

"Generated output" on page 625

#### Related tasks

"Generating from the EGL Software Development Kit (SDK)" on page 418

#### Related reference

"genDirectory" on page 477

"EGLCMD" on page 572

"EGL build path and eglpath" on page 571

"EGLSDK" on page 583

---

## Invoking a build plan after generation

You may wish to create a build plan and to invoke that plan at a later time. This case might occur, for example, if a network failure prevents you from preparing code on a remote machine at generation time.

To invoke a build plan in this case, complete the following steps:

1. Make sure that *eglb*atchgen.jar is in your Java classpath, as happens automatically on the machine where you install EGL. The jar file is in the following directory:

```
installationDir\egl\eclipse\plugins\  
com.ibm.etools.egl.batchgeneration_version
```

*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The installed version of the plugin; for example, 6.0.0

2. Similarly, make sure that your PATH variable includes that directory.
3. From a command line, enter the following command:

```
java com.ibm.etools.egl.distributedbuild.BuildPlanLauncher bp
```

*bp* The fully qualified path of the build plan file. For details on the name of the generated file, see *Generated output (reference)*.

#### Related concepts

"Build plan" on page 413

"Generation" on page 409

#### Related tasks

"Building EGL output" on page 413

#### Related reference

"Build descriptor options" on page 464

"Generated output (reference)" on page 626

---

## Generating Java; miscellaneous topics

### Processing Java code that is generated into a directory

This page describes how to process code that is generated into a directory. It is recommended, however, that you avoid generating code into a directory; for details see *Generation of Java code into a project*.

To generate code into a directory, specify the build descriptor option **genDirectory** and avoid specifying the build descriptor option **genProject**.

Your next tasks depend on the project type:

#### Application client project

For an application client project, do as follows:

1. Provide preparation-time access to EGL jar files by adding the following entries to the project's Java build path:

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar  
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

For details on the variable at the beginning of each entry, see *Setting the variable EGL\_GENERATORS\_PLUGINDIR*.

2. Provide runtime access to fda6.jar, fdaj6.jar, and (if you are calling the generated program by way of TCP/IP) EGLTcpiListener.jar:

- Access the jar files from the following directory:

```
installationDir\egl\eclipse\plugins\  
com.ibm.etools.egl.generators_version\runtime
```

*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The installed version of the plugin; for example, 6.0.0

Copy those files into each enterprise application project that references the application client project.

- Update the manifest in the application client project so that the jar files (as stored in an enterprise application project) are available.
3. Provide access to non-EGL jar files (an optional task)
  4. Import your generated output into the project, in keeping with these rules:
    - The folder *appClientModule* must include the top-level folder of the package that contains your generated output
    - The hierarchy of folder names beneath *appClientModule* must match the name of your Java package

If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

```
appClientModule/my/trial/package
```

5. If you generated a J2EE environment file, update that file



6. Update the deployment descriptor
7. Now that you have placed output files in a project, continue setting up the J2EE runtime environment

### EJB project

For an EJB project, do as follows:

1. Provide preparation-time access to EGL jar files (fda6.jar and fdaj6.jar) by adding the following entries to the project's Java build path:

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

For details on the variable at the beginning of each entry, see *Setting the variable EGL\_GENERATORS\_PLUGINDIR*.

2. Provide runtime access to the EGL jar files:

- Access fda6.jar and fdaj6.jar from the following directory:

```
installationDir\egl\eclipse\plugins\
com.ibm.etools.egl.generators_version\runtime
```

*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The installed version of the plugin; for example, 6.0.0

Copy those files into each enterprise application project that references the EJB project.

- Update the manifest in the EJB project so that fda6.jar and fdaj6.jar (as stored in an enterprise application project) are available.

3. Provide access to non-EGL jar files (an optional task)
4. Import your generated output into the project, in keeping with these rules:
  - The folder *ejbModule* must include the top-level folder of the package that contains your generated output
  - The hierarchy of folder names beneath *ejbModule* must match the name of your Java package

If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

```
ejbModule/my/trial/package
```

5. If you generated a J2EE environment file, update that file.
6. Update the deployment descriptor
7. Set the JNDI name
8. Generate deployment code
9. Now that you have placed output files in a project, continue setting up the J2EE runtime environment

### J2EE Web project

For a Web project, do as follows:

1. Provide access to EGL jar files by copying fda6.jar and fdaj6.jar into your Web project folder. To do so, import the external jars found in the following directory:

```
installationDir\egl\eclipse\plugins\
com.ibm.etools.egl.generators_version\runtime
```



*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The installed version of the plugin; for example, 6.0.0

The destination for the files is the following project folder:

WebContent/WEB-INF/lib

2. Provide access to non-EGL jar files (an option)
3. Import your generated output into the project, in keeping with these rules:
  - The folder *WebContent* must include the top-level folder of the package that contains your generated output
  - The hierarchy of folder names beneath *WebContent* must match the name of your Java package

If you are importing generated output from package *my.trial.package*, for example, you must import that output into a folder that resides in the following location:

WebContent/my/trial/package

4. Update the deployment descriptor
5. Now that you have placed output files in a project, continue setting up the J2EE runtime environment

### Java project

If you are generating code for use in a non-J2EE environment, you generate a properties file if you use the following combination of build descriptor options:

- **genProperties** is set to GLOBAL or PROGRAM; and
- **J2EE** is set to NO.

If you request a global properties file (**rununit.properties**), EGL places that file in the top-level directory. If you request a program properties file instead, EGL places that file with the program, either in the folder that corresponds to the last qualifier in the package name or in the top-level directory. (The top-level directory is used if the package name is not specified in the EGL source file.)

At run time, values in the program properties file are used to set up a standard JDBC connection. For details, see *Understanding how a standard JDBC connection is made*.

For a Java project, your tasks are as follows:

1. Provide access to EGL jar files by adding the following entries to the project's Java build path:

EGL\_GENERATORS\_PLUGIN\_DIR/runtime/fda6.jar  
EGL\_GENERATORS\_PLUGIN\_DIR/runtime/fdaj6.jar

For details on the variable at the beginning of each entry, see *Setting the variable EGL\_GENERATORS\_PLUGIN\_DIR*.

2. If your program accesses a relational database, make sure that your Java build path includes the directory where the driver is installed. For DB2, for example, specify the directory that contains db2java.zip.
3. If your generated code accesses MQSeries, provide access to non-EGL jar files
4. Make sure that the program properties file (if present) is in the top-level project folder and that the global properties file (**rununit.properties**, if

present) is either in the folder that corresponds to the last qualifier in the package name or in the top-level project folder. (The top-level folder is used if the package name is not specified in the EGL source file.)

5. Place a linkage properties file in the project (an optional task)

#### **Related concepts**

"Generation of Java code into a project" on page 409

#### **Related tasks**

"Generating deployment code for EJB projects"

"Deploying a linkage properties file" on page 447

"Setting deployment-descriptor values" on page 438

"Providing access to non-EGL jar files" on page 448

"Setting the JNDI name for EJB projects" on page 441

"Setting the variable EGL\_GENERATORS\_PLUGINDIR"

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

"Understanding how a standard JDBC connection is made" on page 309

"Updating the deployment descriptor manually" on page 441

"Updating the J2EE environment file" on page 440

#### **Related reference**

"genDirectory" on page 477

"genProject" on page 478

## **Generating deployment code for EJB projects**

After you generate into an EJB project and specify the deployment descriptor properties, you can generate the stubs and skeletons that allow for remote access of the EJB:

1. In the Project Explorer, right-click on the project name; then click **Deploy**
2. Follow the directions specified in the help page on Generating EJB deployment code from the workbench

#### **Related tasks**

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

## **Setting the variable EGL\_GENERATORS\_PLUGINDIR**

The workbench classpath variable EGL\_GENERATORS\_PLUGINDIR contains the fully qualified path to the EGL plug-in in the Workbench. The variable is used in the Java build path when you generate an EGL program into a project of type Java, application client, or EJB.

If you encounter a classpath error that refers to EGL\_GENERATORS\_PLUGINDIR, the variable may not be set. The problem occurs, for example, if you check out an EGL-related project from a software configuration management system like Concurrent Versions System (CVS) before you ever work with an EGL part.

You can set the variable by creating an EGL part, by generating EGL code, or by following these steps:

1. Select **Window**, then **Preferences**
2. On the Preferences page, select **Java**, then **Classpath Variables**
3. Select **New...**

4. At the New Variable Entry page, type `EGL_GENERATORS_PLUGINDIR` and specify the following directory:

`installationDir\egl\eclipse\plugins\  
com.ibm.etools.egl.generators_version`

*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The installed version of the plugin; for example, 6.0.0

After you set the variable, rebuild the project.

#### Related concepts

“Generation of Java code into a project” on page 409

#### Related reference

“genProject” on page 478

---

## Running EGL-generated Java code on the local machine

### Starting a basic or text user interface Java application on the local machine

To start an EGL-generated basic (batch) or text user interface (TUI) Java application on your local machine, do the following:

1. Generate Java source code from your EGL source code; for details see *Generating in the workbench*.
2. In the Project Explorer, expand the **JavaSource** folder and select the Java source file for the application you want to run.
3. On the Workbench menu, select **Run > Run As > Java Application**; or on the Workbench toolbar, click the down arrow next to the **Run** button, then select **Run As > Java Application**.

#### Related concepts

“Generation in the workbench” on page 416

#### Related tasks

“Generating in the workbench” on page 414

### Starting a Web application on the local machine

If you are working with an EGL-based Web application that accesses a JNDI data source, you cannot follow the instructions in the current topic unless you previously configured a Web application server. For background information that is specific to WebSphere, see *WebSphere Application Server and EGL*.

To start a Web application, follow these steps:

1. Generate Java source code from your EGL source code; for details see *Generating in the workbench*.
2. In the Project Explorer, expand the **WebContent** and **WEB-INF** folders. Right-click the JSP you want to run, then select **Run > Run on Server**. The Server Selection dialog is displayed.

3. If you have already configured a server for this Web project, select **Choose an existing server**, then select a server from the list. Click **Finish** to start the server, to deploy the application to the server, and to start the application.
4. If you have not configured a server for this Web project, you can proceed as follows, but only if your application does not access a JNDI data source--
  - a. Select **Manually define a server**.
  - b. Specify the host name, which (for the local machine) is **localhost**.
  - c. Select a server type that is similar to the Web application server on which you intend to deploy your application at run time. Choices include **WebSphere v5.1 Test Environment** and **WebSphere v6.0 Server**.
  - d. If you do not intend to change your choices as you work on the current project, select the check box for **Set server as project default**.
  - e. In most cases, you can avoid this step; but if you wish to specify settings that are different from the defaults, click **Next** and make your selections.
  - f. Click **Finish** to start the server, to deploy the application to the server, and to start the application.

#### Related concepts

"Generation in the workbench" on page 416

"WebSphere Application Server and EGL"

"Web support" on page 215

#### Related tasks

"Generating in the workbench" on page 414

### WebSphere Application Server and EGL

When you run or debug an EGL-written, J2EE application in the Workbench, you are likely to use one of these IBM runtime environments:

- WebSphere v5.1 Test Environment, which supports Java servlet version 2.3 (and earlier) and EJB version 2.0 (and earlier)
- WebSphere Application Server v6.0, which supports Java servlet version 2.4 (and earlier) and EJB version 2.1 (and earlier)

If you are debugging or running code that does not use a J2EE data source, the processes for running code in the two environments are similar and require only a few mouse clicks.

If you require access to a J2EE data source, however, the situation is as follows:

- If you are working with the WebSphere v5.1 Test Environment, do the following two steps in any order:
  1. Identify the data source when you define the server configuration.
  2. Make sure that your application refers to the server-configuration entry for that data source.

This second step involves specifying the JNDI name in the deployment descriptor that is specific to your project. You specify the JNDI name in either of these ways--

- When you create the project; or
- When you update the deployment descriptor.

For details on server configuration, see *Configuring WebSphere Application Server v5.x*.

- If you are working with WebSphere Application Server v6.0, do the following two steps in any order:

1. Identify the data source to the server, as is possible in either of these ways--
  - When you update the application deployment descriptor (application.xml), as is recommended; or
  - When you configure the server at the Administrative Console.

For details on updating the application deployment descriptor, see *Setting up a server to test data sources for WebSphere Application Server v6.0*. For details on using the Administrative Console, see *Configuring WebSphere Application Server v6.x*.

2. Make sure that your application refers to the server-configuration entry for that data source.

This second step involves specifying the JNDI name in the deployment descriptor that is specific to your project. You specify the JNDI name in either of these ways--

- When you create the project; or
- When you update the deployment descriptor.

The benefits of updating the application deployment descriptor rather than working at the Administrative Console are as follows:

- You can deploy the enterprise application to any Web application server that supports J2EE version 1.4, with no additional server configuration necessary for identifying the data source.
- You can update the application deployment descriptor regardless of whether the server is running.
- You gain convenience because your actions are within the development component of your Rational Developer product rather than within the WebSphere Application Server component.

However you update the data-source information, your change is available to the server almost immediately.

#### Related concepts

“Web support” on page 215

---

## Build script

A build script is a file that is invoked by a build plan and that prepares output from generated files. Examples are as follows:

- A Java compiler or other .exe (binary) file or a .bat (text) file is available to a build server on the development system or is sent to a build server on a remote Windows 2000/NT/XP.
- A script (.scr file) or some binary code is sent to a USS build server.

You specify the address of a build machine by setting the build descriptor option **destHost**.

## Java build script

To prepare Java code for execution, EGL puts the javac (Java compiler) command and its parameters in the build plan and sends to the build machine the javac command and the input required by the command.

#### Related concepts

“Build” on page 411

“Build plan” on page 413

“Build server” on page 427

#### Related reference

"Build descriptor options" on page 464

"destDirectory" on page 473

"destHost" on page 474

"destPassword" on page 475

"destUserID" on page 475

"Output of Java program generation" on page 781

"Output of Java wrapper generation" on page 782

---

## Build server

A build server receives requests from a client system to create executable files from source code sent from that client. A build server must be started prior to sending any requests from a build client. A build server typically services requests from multiple clients. Multiple threads may be started if concurrent build requests are received.

In a generator environment you start a build server on a machine whose operating system is the target generation system, for example, Windows 2000. The generator produces Java source code. Java code is sent to a specified build server where the Java compiler is invoked.

If you are generating Java code for Windows, you can build the Java outputs on the same machine as the machine where generation was performed. This is called a local build. In this case you do not have to start a build server. If you want to perform a local build, omit the **destHost** option from the build descriptor.

#### Related concepts

"Build" on page 411

"Build script" on page 426

#### Related tasks

"Starting a build server on AIX, Linux, or Windows 2000/NT/XP"

#### Related reference

"Build descriptor options" on page 464

## Starting a build server on AIX, Linux, or Windows 2000/NT/XP

To start a remote build server on AIX, Linux, or Windows 2000/NT/XP, enter the `ccublds` command in a Command Prompt window. The syntax is as follows:

```
►► ccublds -p portno -V -a 0 2 ►►
```

where

- p** Specifies the port number (*portno*) that the server listens to, to communicate with the clients.
- V** Specifies the verbosity level of the server. You may specify this parameter up to three times (maximum verbosity).
- a** Specifies the authentication mode:

- 0 The server performs builds requested by any client. This mode is recommended only in an environment where security is not a concern.
- 2 The server requires the client to provide a valid user ID and password before accepting a build. The user ID and password are first configured by the owner of the host machine where the build server runs. You do the configuration by using the Security Manager described below.

## Setting the language of messages returned from the build server

The build server on Windows returns messages in any of the languages listed in the next table, and the default is English.

Language	Code
Brazilian Portugese	ptb
Chinese, simplified	chs
Chinese, traditional	cht
English, USA	enu
French	fra
German	deu
Italian	ita
Japanese	jpn
Korean	kor
Spanish	esp

To specify a language other than English, make sure that before you start the build server, the environment variable CCU\_CATALOG is set to a non-English message catalog. The needed value is in the following format (on a single line):

```
installationDir\egl\eclipse\plugins  
\com.ibm.etools.egl.distributedbuild\executables  
\ccu.cat.xxx
```

*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*xxx*

The language code being supported by the build server; one of the codes listed in the previous table

## Security Manager

The Security Manager is a server program that the build server uses to authenticate clients that send build requests.

**Setting the environment for the Security Manager:** The Security Manager uses the following Windows environment variables:

### CCUSEC\_PORT

Sets the number of the port to which the Security Manager listens. The default value is 22825.

## CCUSEC\_CONFIG

Sets the path name of the file in which configuration data is saved. The default is C:\temp\ccuconfig.bin. If this file is not found, the Security Manager creates it.

## CCU\_TRACE

Initiates tracing of the Security Manager for diagnostics purposes, if this variable is set to \*.

**Starting the Security Manager:** To start the Security Manager, issue the following command:

```
java com.ibm.etools.egl.distributedbuild.security.CcuSecManager
```

**Configuring the Security Manager:** To configure the Security Manager, use the Configuration Tool, which has a graphical interface. You can run the tool by issuing the following command:

```
java com.ibm.etools.egl.distributedbuild.security.CCUconfig
```

When Configuration Tool is running, select the **Server Items** tab. Using the button 'Add...', To add the user that you want the build server to support, click the **Add ...** button. You must define a password for the user ID. You can define the following restrictions and privileges for the user:

- The locations, that is, the values of the -la parameter to ccubldc command, that this user can specify. Different locations are separated by semicolons.
- The name of the build script that this user can specify. (The EGL build plan only uses the javac command as a build script.)
- Whether or not this user can send build scripts from client, that is, use the -ft parameter of ccubldc command. (The EGL generator does not use the -ft parameter. You would specify this parameter if they were using the build for purposes other than preparing Java-generation outputs.)

These definitions are kept in persistent storage, in the file specified by CCUSEC\_CONFIG, and are remembered across sessions.

## Related concepts

"Build script" on page 426

"Build server" on page 427

## Related tasks

"Syntax diagram for EGL statements and commands" on page 884





---

## Deploying EGL-generated Java output

---

### Java runtime properties

An EGL-generated Java program uses a set of runtime properties that provide information such as how to access the databases and files that are used by the program.

For details on the runtime properties used by a service generated with EGL, see *Library part of type ServiceBindingLibrary*.

### In a J2EE environment

In relation to a generated Java program that will run in a J2EE environment, these situations are possible:

- EGL can generate the runtime properties directly into a J2EE deployment descriptor. In this case, EGL overwrites properties that already exist and appends properties that do not exist. The program accesses the J2EE deployment descriptor at run time.
- Alternatively, EGL can generate the runtime properties into a J2EE environment file. You can customize the properties in that file, then copy them into the J2EE deployment descriptor.
- You can avoid generating the runtime properties at all, in which case you must write any needed properties by hand.

In a J2EE module, every program has the same runtime properties because all code in the module shares the same deployment descriptor.

In WebSphere Application Server, properties are specified as `env-entry` tags in the `web.xml` file that is associated with the Web project, as in these examples:

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-value>ENU</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>

<env-entry>
  <env-entry-name>vgj.nls.number.decimal</env-entry-name>
  <env-entry-value>.</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

### In a non-J2EE Java environment

In relation to a generated Java program that runs outside of a J2EE environment, you can generate the runtime properties into a program properties file or code that file by hand. (The program properties file provides the kind of information that is available in the deployment descriptor, but the format of the properties is different.)

In a non-J2EE Java environment, properties can be specified in any of several properties files, which are searched in this order:

- **user.properties**
- A file named as follows--

*programName.properties*

*programName*

The first program in the run unit

- **rununit.properties**

Use of **user.properties** is appropriate when you specify properties that are specific to a user. EGL does not generate content for this file.

Use of **rununit.properties** is especially appropriate when the first program of a run unit does not access a file or database but calls programs that do:

- When generating the caller, you can generate a properties file named for the program, and the content might include no database- or file-related properties
- When you generate the called program, you can generate **rununit.properties**, and the content would be available for both programs

None of those files is mandatory, and simple programs do not need any.

At deployment time, these rules apply:

- The user properties file ( **user.properties**, if present) is in the user home directory, as determined by the Java system property *user.home*.
- The location of a program properties file (if present) depends on whether the program is in a package. The rules are best illustrated by example:
  - If program P is in package x.y.z and is deployed to MyProject/JavaSource, the program properties file must be in MyProject/JavaSource/x/y/z
  - If program P is not in a package and is deployed to myProject/JavaSource, the program properties file (like the global properties file) must be in MyProject/JavaSource

In either case, MyProject/JavaSource must be in the classpath.

- The global properties file (**rununit.properties**, if present) must be with the program, in a directory that is specified in the classpath.

If you generate output to a Java project, EGL places the properties files (other than **user.properties**) in the appropriate folders.

If you are generating Java code for use in the same run-unit as Java code generated with an earlier version of EGL or VisualAge Generator, the rules for deploying properties file depends on whether the first program in the run unit was generated with EGL 6.0 or later (in which case the rules described here apply) or whether the first program was generated with an earlier version of EGL or VisualAge Generator (in which case the properties files can be in any directory in the classpath, and the global file is called **vgj.properties**).

Finally, if the first program was generated with the earlier software, you can specify an alternate properties file, which is used throughout the run unit in place of any non-global program properties files. For details, see the description of property **vgj.properties.file** in *Java runtime properties (details)*.

## Build descriptors and program properties

Choices are submitted to EGL as build-descriptor-option values:

- To generate properties into a J2EE deployment descriptor, set **J2EE** to YES; set **genProperties** to PROGRAM or GLOBAL; and generate into a J2EE project.

- To generate properties into a J2EE environment file, set **J2EE** to YES; set **genProperties** to PROGRAM or GLOBAL; and do either of these:
  - Generate into a directory (in which case you use the build descriptor option **genDirectory** rather than **genProject**); or
  - Generate into a non-J2EE project.
- To generate a program properties file with the same name as the program being generated, set **J2EE** to NO; set **genProperties** to PROGRAM; and generate into a project other than a J2EE project.
- To generate a program properties file **rununit.properties**, set **J2EE** to NO; set **genProperties** to GLOBAL; and generate into a project other than a J2EE project.
- To avoid generating properties, set **genProperties** to NO.

## For additional information

For details on generating properties into a deployment descriptor or into a J2EE environment file, see *Setting deployment-descriptor values*.

For details on the meaning of the runtime properties, see *Java runtime properties (details)*.

For details on accessing runtime properties in your EGL code, see *sysLib.getProperty*.

### Related concepts

“EGL debugger” on page 369  
 “Generation of Java code into a project” on page 409  
 “J2EE environment file” on page 440  
 “Library part of type ServiceBindingLibrary” on page 172  
 “Program properties file”  
 “Run unit” on page 866

### Related tasks

“Processing Java code that is generated into a directory” on page 420  
 “Setting up the J2EE runtime environment for EGL-generated code” on page 437  
 “Setting deployment-descriptor values” on page 438  
 “Updating the deployment descriptor manually” on page 441  
 “Updating the J2EE environment file” on page 440

### Related reference

“genProperties” on page 480  
 “J2EE” on page 483  
 “Java runtime properties (details)” on page 642  
 “getProperty()” on page 1033

---

## Setting up the non-J2EE runtime environment for EGL-generated code

### Program properties file

The *program properties file* contains Java runtime properties in a format that is accessible only to a Java program that runs outside of a J2EE environment. For overview information, see *Java runtime properties*.

The program properties file is a text file. Each entry other than comments has the following format:

```
propertyName = propertyValue
```

*propertyName*

One of the properties described in *Java runtime properties (details)*

*propertyValue*

The property value that is available to your program at run time

A comment is any line where the first non-text character is a pound sign (#).

A portion of an example file is as follows:

```
# This file contains properties for generated  
# Java programs that are being debugged in a  
# non-J2EE Java project
```

```
vgj.nls.code = ENU  
vgj.datemask.gregorian.long.ENU = MM/dd/yyyy
```

For details on the name given to the generated file, see *Generated output (reference)*.

#### **Related concepts**

“EGL debugger” on page 369

“Java runtime properties” on page 431

#### **Related tasks**

“Generated output (reference)” on page 626

#### **Related reference**

“genProperties” on page 480

“J2EE” on page 483

“Java runtime properties (details)” on page 642

## **Deploying Java applications outside of J2EE**

To deploy a Java application outside of J2EE, do as follows:

1. Follow the procedure detailed in *Installing the EGL runtime code for Java*
2. Export the EGL-generated code into jar files, remembering to include generated output files that have extensions other than java; for example, jasper, properties, and tab files
3. Export any manually written Java code into jar files
4. Include the exported files in the classpath of the target machine

#### **Related tasks**

“Installing the EGL runtime code for Java”

## **Installing the EGL runtime code for Java**

The EGL runtime code for generated Java applications is available in a zip file on the following Web site:

<http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rad/60/redist>

The supported distributed platforms are AIX, HP-UX, Linux (Intel®), iSeries, Solaris, and Windows 2000/NT/XP. (See product prerequisites for supported versions.) EGL provides 32- and 64-bit support for AIX, HP-UX, and Solaris.

The zip file you download from the previously mentioned Web site includes the following:

- Jar files which contain Java code that is common to all supported distributed platforms
- Platform-specific code

Do as follows:

1. Extract the files in the EGLRuntimes directory to each machine on which deployed EGL applications are to be run outside of a J2EE application server. (These files are already included in any Enterprise Archive (EAR) file used to deploy J2EE applications.)
2. Include the jar files in the classpath of the deployment machines.
3. Copy any platform-specific code to a directory on each deployment machine; and set environment variables for each of those machines as appropriate:

**For AIX (32- or 64-bit support)**

The files of interest are in the directory **Aix** or (for 64-bit support) **Aix64**. Change the PATH and LIBPATH environment variables so they reference the directory that contains the platform-specific code you copied from the Web site.

**For HP-UX (32- or 64-bit support)**

The files of interest are in the directory **hpux** or (for 64-bit support) **hpux64**. Change the PATH and LIBPATH environment variables so they reference the directory that contains the platform-specific code you copied from the Web site.

**For iSeries**

The files of interest are in the directory **Iseries**. In qshell, change to the directory you just uploaded the files to and run the setup.sh script with the "install" option:

```
> setup.sh install
```

In addition, some other environment variables must be set. For information on how to set these environment variables, run the script with the "envinfo" option:

```
> setup.sh envinfo
```

If for some reason you delete a symlink that is created for you during install, you can recreate it with the "link" option:

```
> setup.sh link
```

**For Linux**

The files of interest are in the directory **Linux**. Change the PATH and LIBPATH environment variables so they reference the directory that contains the platform-specific code you copied from the Web site.

**For Solaris (32- or 64-bit support)**

The files of interest are in the directory **Solaris** or (for 64-bit support) **Solaris64**. Change the PATH and LIBPATH environment variables so they reference the directory that contains the platform-specific code you copied from the Web site.

**For Windows 2000/NT/XP**

The files of interest are in the directory **Win32**. Change the PATH environment variable so it references the directory that contains the platform-specific code you copied from the Web site.

#### Related tasks

“Deploying Java applications outside of J2EE” on page 434

## Including JAR files in the CLASSPATH of the target machine

You must include any JAR files that contain EGL-generated code or manually written Java code in the CLASSPATH of the target machine. The steps for this process are system dependent. See your operating system documentation for details.

## Setting up the UNIX curses library for the EGL runtime

When you deploy an EGL text program on AIX or Linux, the EGL runtime tries to use the UNIX curses library. If the environment is not set up for the UNIX curses library or if that library is not supported, the EGL runtime tries to use the Java Swing technology; and if that technology is also not available, the program fails.

The UNIX curses library is required when the user runs an EGL program from a terminal emulator window or a character terminal.

To enable the EGL runtime to access the UNIX curses terminal library on AIX or Linux, you must fulfill several steps in the UNIX shell environment. In each of the first two steps, *installDir* refers to the runtime installation library:

1. Modify the LD\_LIBRARY\_PATH environment variable to include the shared object libCursesCanvas6.so, which is provided in the runtime installation library--

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: /installDir/bin
```

2. Modify the CLASSPATH environment variable to add fda6.jar and fdaj6.jar--  
export CLASSPATH=\$CLASSPATH:  
/installDir/lib/fda6.jar: /installDir/lib/fdaj6.jar

The previous information must be typed on a single line.

3. Set the TERM environment variable to the appropriate terminal setting, as in the following example:

```
export TERM=vt100
```

If terminal exceptions occur, try various terminal settings such as xterm, dtterm, or vt220.

4. Run your EGL Java program from the UNIX shell, as in the following example:

```
java myProgram
```

Make sure that the CLASSPATH environment variable identifies the directory in which your program resides.

For additional details on using the Curses library on UNIX, refer to the UNIX man pages.

## Setting up the TCP/IP listener for a called non-J2EE application

If you want a caller to use TCP/IP to exchange data with a called non-J2EE Java program, you must set up a TCP/IP listener for the called program.

If you are using TCP/IP to communicate with a called non-J2EE Java program, you must configure a standalone Java program called CSOTcpipListener for that program. Specifically, you must do as follows:

- Make sure that the classpath used when running CSOTcpipListener contains fda6.jar, fdaj6.jar, and the directories or archives that contain the called programs; and
- Set the Java runtime property **tcpiplistener.port** to the number of the port at which CSOTcpipListener receives data.

You can start the standalone TCP/IP listener in either of two ways:

- To start the listener from the workbench, use the launch configuration for a Java application. In this case, you can specify the name of the properties file in the program arguments of the launch configuration. Alternatively, if you are using the file tcpiplistener.properties as a default, that file should not be in a folder, but should be directly under the project that you specified when you created the launch configuration.

- To start the listener from the command line, run the program as follows:

```
java CSOTcpipListener propertiesFile
```

*propertiesFile*

The fully qualified path to the properties file used by the TCP/IP listener. If you do not specify a properties file, the listener attempts to open the following file in the current directory:

```
tcpiplistener.properties
```

#### Related tasks

“Providing access to non-EGL jar files” on page 448

---

## Setting up the J2EE runtime environment for EGL-generated code

EGL-generated Java programs and wrappers run on a J2EE 1.4 server such as WebSphere Application Server v6.0, on the platforms listed in *Runtime configurations*.

Your primary tasks when embedding generated Java classes into a J2EE module are as follows:

1. Place output files in a project, in either of two ways:
  - Generate into a project, as is the preferred technique; or
  - Generate into a directory, then import files into a project.
2. Place a linkage properties file in the module (see *Deploying a linkage properties file*).
3. Eliminate duplicate jar files.
4. Export an enterprise archive (.ear) file, which may include Web application archive (.war) files and other .ear files; for details on the procedure, see the help pages on export.
5. Import the .ear file into the J2EE server that will host your application; for details on the procedure, see the documentation for your J2EE server.

You may need to fulfill these tasks as well:

- “Setting up a J2EE JDBC connection” on page 445
- “Setting up the J2EE server for CICSJ2C calls” on page 441
- “Setting up the J2EE server for IMSJ2C calls” on page 442
- “Setting up the TCP/IP listener for a called appl in a J2EE appl client module” on page 442
- “Setting up the TCP/IP listener for a called non-J2EE application” on page 436



### Related concepts

- "Development process" on page 9
- "Generation of Java code into a project" on page 409
- "Java program, PageHandler, and library" on page 414
- "Linkage options part" on page 399
- "Linkage properties file" on page 447
- "Runtime configurations" on page 11

### Related tasks

- "Deploying Java applications outside of J2EE" on page 434
- "Deploying a linkage properties file" on page 447
- "Eliminating duplicate jar files"
- "Generating deployment code for EJB projects" on page 423
- "Processing Java code that is generated into a directory" on page 420
- "Providing access to non-EGL jar files" on page 448
- "Setting deployment-descriptor values"
- "Setting the JNDI name for EJB projects" on page 441
- "Setting up a J2EE JDBC connection" on page 445
- "Setting up the J2EE server for CICSJ2C calls" on page 441
- "Setting up the TCP/IP listener for a called appl in a J2EE appl client module" on page 442
- "Understanding how a standard JDBC connection is made" on page 309
- "Updating the deployment descriptor manually" on page 441
- "Updating the J2EE environment file" on page 440

### Related reference

- "Java runtime properties (details)" on page 642
- "Linkage properties file (details)" on page 764

## Eliminating duplicate jar files

If you place multiple J2EE modules into a single ear file, eliminate the duplicate jar files as follows:

1. Move a copy of each duplicate jar file to the top level of the ear
2. Delete the duplicate jar files from the J2EE modules
3. Ensure that the build path for each of the affected J2EE modules points to the jar files in the ear; specifically, do as follows for each of those J2EE modules:
  - a. Right-click on the module from within the Project Explorer or J2EE view
  - b. Select **Edit Module Dependencies**
  - c. When the Module Dependencies dialog is displayed, select the jar files to access from the top level of the ear, then click **Finish**.

### Related tasks

- "Setting up the J2EE runtime environment for EGL-generated code" on page 437

## Setting deployment-descriptor values

An important task is to place runtime values (similar to environment-variable values) into the deployment descriptor of your J2EE module. You can interact with a workbench editor listed in the next table, for example; and in any case, the editors are available if you wish to reassign a value.

Project type	Name of deployment descriptor	How to assign values
application client	application-client.xml	Use the XML editor, Design tab

Project type	Name of deployment descriptor	How to assign values
EJB	ejb-jar.xml	Use the EJB editor, Beans tab
J2EE Web	web.xml	Use the web.xml editor, Environment tab

The recommended way to update the deployment descriptor is to add content automatically, as happens if all of the following conditions apply:

- You are generating a Java program or wrapper
- The build descriptor option **genProperties** is set to GLOBAL or PROGRAM
- You are generating for J2EE run time by setting **J2EE** to YES
- You set **genProject** to a valid J2EE project

EGL never deletes a property from an existing deployment descriptor, but does as follows:

- Overwrites properties that already exist
- Appends properties that do not exist

Another method of updating the deployment descriptor is to paste values from the J2EE environment file, which is an output of generation if all of the following conditions apply:

- You are generating a Java program
- The build descriptor option **genProperties** is set to GLOBAL or PROGRAM
- You are generating for J2EE run time by setting **J2EE** to YES
- You do not set **genProject** to a valid J2EE project, as when you generate into a directory instead

Before you paste entries from a J2EE environment file into the deployment descriptor of an application client or EJB project, you need to change the order of entries in the file, as described in *Updating the J2EE environment file*. You do not need to change the order of entries if you are working with a J2EE Web project.

For details on deployment descriptor properties, see *Java runtime properties (details)*.

### Related concepts

"J2EE environment file" on page 440

"Generation of Java code into a project" on page 409

"Program properties file" on page 433

### Related tasks

"Processing Java code that is generated into a directory" on page 420

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

"Updating the J2EE environment file" on page 440

"Updating the deployment descriptor manually" on page 441

### Related reference

"genDirectory" on page 477

"genProperties" on page 480

"J2EE" on page 483

"Java runtime properties (details)" on page 642

## Updating the J2EE environment file

The J2EE environment file contains a series of entries like the following example:

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-value>ENU</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

The order of sub-elements is name, value, type. This is correct for J2EE Web projects; however, for application client and EJB projects, you need to change the order to name, type, value. For the example above, change the order of the sub-elements to:

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>ENU</env-entry-value>
</env-entry>
```

This step can be avoided if you generate directly into a project instead of into a directory. When you generate into a project, EGL can determine the type of project you are using and generate the environment entries in the appropriate order.

### Related tasks

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

“Setting deployment-descriptor values” on page 438

### Related reference

“Java runtime properties (details)” on page 642

## J2EE environment file

A *J2EE environment file* is a text file that contains property-and-value pairs that are derived from information that you specify when you generate a Java program. The sources of information are the build descriptor, the resource associations part, and the linkage options part.

When you configure the environment of the Java program, you can use the J2EE environment file as the basis of information that you place in the runtime deployment descriptor.

For details on the name of the J2EE environment file, see *Generated output (reference)*.

For details on the different ways that you can set deployment-descriptor values, see *Setting deployment-descriptor values*.

### Related concepts

“Runtime configurations” on page 11

### Related tasks

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

“Setting deployment-descriptor values” on page 438

### Related reference

“Generated output (reference)” on page 626

“genProperties” on page 480

“sqlDB” on page 490

## Updating the deployment descriptor manually

If you are updating a deployment descriptor from a generated J2EE environment file, do as follows:

1. Read the overview information in *Setting deployment-descriptor values*.
2. If you worked on an application client or EJB project, you must make sure that the order of the sub-elements in the generated environment entries is correct, as described in *Updating the J2EE environment file*.
3. Copy the environment entries into your project's deployment descriptor as follows:
  - a. Make a backup copy of the deployment descriptor.
  - b. Open the J2EE environment file, which is called *programName-env.txt* file. Copy the environment entries into the clipboard.
  - c. Double-click on the deployment descriptor.
  - d. Click on the Source tab.
  - e. Paste the entries at a proper location.

For details on deployment descriptors, see *Java runtime properties (details)*.

### Related tasks

"Setting deployment-descriptor values" on page 438

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

"Updating the J2EE environment file" on page 440

### Related reference

"Java runtime properties (details)" on page 642

## Setting the JNDI name for EJB projects

To set the JNDI name for an EJB project, do as follows:

1. Right click on *ejb-jar.xml* (the deployment descriptor) to open the context menu.
2. Use the EJB Editor to open the following file in the project--  
`\ejbModule\META-INF\ejb-jar.xml`
3. Click on the Beans tab.
4. On the list, click on the name of the EJB you just generated.
5. Enter the JNDI name under WebSphere Bindings. The JNDI name must be as follows for use by the EGL runtime code:
  - First character of the program name, in upper case
  - Subsequent characters of the program name, in lower case
  - The letters EJB in upper case.

### Related tasks

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

## Setting up the J2EE server for CICSJ2C calls

You must set up a *ConnectionFactory* in the J2EE server for each CICS transaction accessed through protocol CICSJ2C.

If a generated Java wrapper is making the CICSJ2C call, you can handle security in any of the following ways (where a wrapper-specified value overrides that of the J2EE server):

- Set the userid and password in the wrapper's CSOCallOptions object; or
- Set the userid and password in the ConnectionFactory configuration in the J2EE server; or
- Set up the CICS region so that user authentication is not required.

When calling a program from WebSphere 390, the following restrictions apply:

- If the callLink element property **luwControl** is set to CLIENT, the call fails. The WebSphere 390 connect implementation does not support an extended unit of work.
- The setting of deployment descriptor property **cso.cicsj2c.timeout** has no effect. By default, timeouts never occur. In the EXCI options table generated by the macro DFHXCOPT, however, you can set the parameter TIMEOUT, which lets you specify the time that EXCI will wait for a DPL command (an ECI request) to complete. A setting of 0 means to wait indefinitely.

For details, see *Java Connectors for CICS: Featuring the J2EE Connector Architecture* (SG24-6401-00), which is available from web site <http://www.redbooks.ibm.com>.

#### Related tasks

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

## Setting up the J2EE server for IMSJ2C calls

You must set up a ConnectionFactory in the J2EE server for each IMS transaction accessed through protocol IMSJ2C.

If a generated Java wrapper is making the IMSJ2C call, you can handle security in any of the following ways (where a wrapper-specified value overrides that of the J2EE server):

- Set the userid and password in the wrapper's CSOCallOptions object; or
- Set the userid and password in the ConnectionFactory configuration in the J2EE server; or
- Set up the IMS region so that user authentication is not required.

If the callLink element property **luwControl** is set to CLIENT when you are calling a program from WebSphere 390, the call fails. The WebSphere 390 connect implementation does not support an extended unit of work.

For other details, see *e-business Cookbook for z/OS Volume II: Infrastructure* (SG24-5981-01), which is available from Web site <http://www.redbooks.ibm.com>.

#### Related tasks

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

## Setting up the TCP/IP listener for a called appl in a J2EE appl client module

If you want a caller to use TCP/IP to exchange data with a called program in a J2EE application client module, you must set up a TCP/IP listener for the called program.

You need to make sure that the following situation is in effect:

- An EGL-specific TCP/IP listener is the main class for the module, as specified in the manifest (.MF) file of the module

- A port is assigned to the listener, as specified in the deployment descriptor (application-client.xml) of the module

If you are working with projects at the level of J2EE 1.2, it is recommended that you set up an application client project that is initialized with the listener, before you generate any EGL code into that project. If you fail to follow that sequence (listener first, EGL code second) or if you are working with projects at the level of J2EE 1.3, you need to follow the procedure described in *Providing access to the listener from an existing application client project*.

## Setting up an application client project that is initialized with the listener

To set up an application client project that is initialized with the listener, do as follows:

1. Click **File > Import**.
2. At the Select page, double-click **App Client JAR file**.
3. At the Application Client Import page, specify several details--
  - a. In the Application Client file field, specify the jar file that sets up access to (but does not include) the TCP/IP listener:  
`installationDir\egl\eclipse\plugins\com.ibm.etools.egl.generators_version\runtime\EGLTcpListener.jar`  
*installationDir*  
 The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.  
*version*  
 The latest version of the plugin; for example, 6.0.0.  
 The TCP/IP listener itself resides in fdaj6.jar, which is placed in the application client project when you first generate EGL code into that project.
  - b. Click the **New** radio button, which follows the label **Application Client project**.
  - c. Type the name of the application client project into the **New Project Name** field; then set or unset the **Use default** check box. If you set the check box, the project is stored in a workspace directory that is named with the name of the project. If you unset the check box, specify the project name in the **New project location** field.
  - d. Specify the name of the enterprise application project that contains the application client project:
    - If you are using an existing J2EE 1.2 enterprise application project, click the **Existing** radio button, which follows the label **Enterprise application project**. In this case, specify the project name in the **Existing project name** field.
    - If you are creating a new enterprise application project, do as follows:
      - 1) Click the **New** radio button, which follows the label **Enterprise application project**.
      - 2) Type the name of the enterprise application project into the **New Project Name** field.
      - 3) Set or unset the **Use default** check box.
      - 4) If you set the check box, the project is stored in a workspace directory that is named with the name of the project. If you unset the check box, specify the project name in the **New project location** field.

4. Click **Finish**.
5. Ignore the two warning messages that refer to the jar files (fda6.jar, fdaj6.jar) that will be added automatically when you generate EGL output into the project.

In the application client project, the deployment descriptor property **tcpiplistener.port** is set to the number of the port at which the listener receives data. By default, that port number is 9876. To change the port number, do as follows:

1. In the Project Explorer view, expand your application client project, then appClientModule, then META-INF
2. Click **application-client.xml > Open With > Deployment Descriptor editor**
3. The deployment descriptor editor includes a source tab; click that tab and change the 9876 value, which is the content of the last tag in a grouping like this:
 

```
<env-entry-name>tcpiplistener.port</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-name>
<env-entry-value>9876</env-entry-value>
```
4. To save the deployment descriptor, press **Ctrl-S**.

### Providing access to the listener from an existing application client project

If you generate EGL code into an application client project that was not initialized with the listener, you need to update the deployment descriptor (application-client.xml) and the manifest file (MANIFEST.MF):

1. In the Project Explorer view, expand your application client project, then appClientModule, then META-INF
2. Click **application-client.xml > Open With > Deployment Descriptor Editor**
3. The deployment descriptor editor includes a Source tab. Click that tab. In the text, immediately below the line that holds the tag <display-name>, add the following entries (however, if port 9876 is already in use on your machine, substitute a different number for 9876):

```
<env-entry>
  <env-entry-name>tcpiplistener.port</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-name>
  <env-entry-value>9876</env-entry-value>
</env-entry>
```

4. To save the deployment descriptor, press **Ctrl-S**.
5. In the Project Explorer view, click **MANIFEST.MF > Open With > JAR Dependency Editor**.
6. The JAR Dependency Editor includes a Dependencies tab. Click that tab.
7. Review the Dependencies section to make sure that fda6.jar and fdaj6.jar are selected.
8. In the Main Class section, in the Main-Class field, type the following value or use the Browse mechanism to specify the following value:
 

```
CS0TcpiListenerJ2EE
```
9. To save the manifest file, press **Ctrl-S**.

### Deploying the application client project

To start the TCP/IP listener, follow either of two procedures:

- Start the listener from the Workbench by using the launch configuration for a WebSphere application client:
  1. Switch to a J2EE perspective



2. Click **Run > Run**
3. At the Launch Configurations page, click either **WebSphere v5 Application Client** (as is necessary if you are working with a project at the level of J2EE 1.3) or **WebSphere v4 Application Client**
4. Select an existing configuration. Alternatively, click **New** and set up a configuration:
  - a. In the Application tab, select the enterprise application project
  - b. In the Arguments tab, add an argument:

`-CCjar=myJar.jar`

*myJar.jar*

The name of the application client jar file. This argument is only necessary when you have multiple client jar files in the ear file. In most cases, the value is the name of the application client project, followed by the extension .jar.

If you wish to confirm the relationship of project name to jar-file name, do as follows:

- 1) In the Project Explorer view, expand your enterprise application project, then META-INF
  - 2) Click **application.xml > Open With > Deployment Descriptor Editor**.
  - 3) The Deployment Descriptor editor includes a Module tab. Click that tab.
  - 4) At the leftmost part of the page, click the jar file and see (at the rightmost part of the page) the project name associated with that jar file.
- If you have on WebSphere Application Server (WAS) installed, you can use launchClient.bat, which is in the WAS installation directory, subdirectory bin. You can invoke launchClient as follows from a command prompt:

```
launchClient myCode.ear -CCjar=myJar.jar
```

*myCode.ear*

The name of the enterprise archive

*myJar.jar*

The name of the application client jar file, as described in relation to the Workbench procedure

For details on launchClient.bat, see the WebSphere Application Server documentation.

#### Related tasks

"Providing access to non-EGL jar files" on page 448

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

## Setting up a J2EE JDBC connection

If you are connecting to a relational database at run time, you need to define a data source for use with your program. The directions are in the help system of the WebSphere Server administrative console.

When you define a data source, assign values to the following properties:

#### JNDI name

Specify a value that matches the name to which the database is bound in the JNDI registry:



- If you are defining a data source which connects to a database that your J2EE module uses by default, make sure that the JNDI name specified in the data source definition matches the value of the **vgj.jdbc.default.database** property in the J2EE deployment descriptor used at run time
- If you are defining a data source that will be accessed when the system function `VGLib.connectionService` runs, make sure that the JNDI name specified in the data source definition matches the value of the appropriate **vgj.jdbc.database.SN** property in the J2EE deployment descriptor used at run time

#### Database name

Specify the name of your database, as known to the database management system

#### User ID

Specify the user name for connecting to the database.

If the data source definition refers to the default database, the value you specify in the User ID field is overridden by any value set in the **vgj.jdbc.default.userid** property of the J2EE deployment descriptor used at run time, but only if you have specified values for both **vgj.jdbc.default.userid** and **vgj.jdbc.default.password**. Similarly, if the data source definition refers to a database that is accessed by way of the system function `sysLib.connect` or `VGLib.connectionService`, the value you specify in the User ID field is overridden by any user ID that you specify in the call to that system function, but only if the call passes both a user ID and password.

You specify the name when setting up the authentication alias. To reach the display where you can define that alias, follow this sequence in the Administrative Console: **Security > GlobalSecurity > Authentication > JAAS Configuration > J2C Authentication Data**.

#### Password

Specify the password for connecting to the database. If the data source definition refers to the default database, the value you specify in the Password field is overridden by any value set in the **vgj.jdbc.default.password** property of the J2EE deployment descriptor used at run time, but only you have specified values for both **vgj.jdbc.default.userid** and **vgj.jdbc.default.password**. Similarly, if the data source definition refers to a database that is accessed by way of the system function `VGLib.connectionService`, the value you specify in the Password field is overridden by any password that you specify in the call to that system function, but only if the call passes both a user ID and password.

You specify the password when setting up the authentication alias. To reach the display where you can define that alias, follow this sequence in the Administrative Console: **Security > GlobalSecurity > Authentication > JAAS Configuration > J2C Authentication Data**.

You may define multiple data sources, in which case you use the system function `VGLib.connectionService` to switch between them.

For details on the meaning of the deployment descriptor properties, including details on how the generated values are derived, see *Java runtime properties (reference)*.

#### Related tasks

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

“Understanding how a standard JDBC connection is made” on page 309

**Related reference**

"Java runtime properties (details)" on page 642  
"JDBC driver requirements in EGL" on page 660  
"connectionService()" on page 1047

## Deploying a linkage properties file

The linkage properties file must be in the same J2EE application as the Java program that uses the file. If the file is in the top-level directory of the application, set the Java runtime property **cso.linkageOptions.LO** to the file name, without path information. If the file is under the top-level directory of the application, use a path that starts at the top-level directory and includes a virgule (/) for each level, even if the application is running on a Windows platform.

When you are developing a J2EE project, the top-level directory corresponds to the appClientModule, ejbModule, or Web Content directory of the project in which the module resides. When you are developing a Java project, the top-level directory is the project directory.

For additional details on how a linkage properties file is formatted and identified, see *Linkage properties file (reference)*.

**Related concepts**

"Java runtime properties" on page 431  
"Linkage options part" on page 399  
"Linkage properties file"

**Related tasks**

"Setting up the J2EE runtime environment for EGL-generated code" on page 437  
"Setting deployment-descriptor values" on page 438

**Related reference**

"callLink element" on page 499  
"Exception handling" on page 94  
"Linkage properties file (details)" on page 764

## Linkage properties file

A *linkage properties file* is a text file that is used at Java run time to give details on how a generated Java program or wrapper calls a generated Java program in a different process.

The file is applicable only if you specified that linkage options for a Java program or wrapper are set at run time instead of at generation time. You may generate the file or create one from scratch.

For details on when the file is generated and on the file format, see *Linkage properties file (details)*. For details on the name of the generated file, see *Generated output (reference)*. For details on deployment, see *Deploying a linkage properties file*.

**Related concepts**

"Generated output" on page 625

**Related tasks**

"Deploying a linkage properties file"  
"Setting up the J2EE runtime environment for EGL-generated code" on page 437

#### Related reference

“Generated output (reference)” on page 626

“genProperties” on page 480

“Linkage properties file (details)” on page 764

## Providing access to non-EGL jar files

You may need to provide access to non-EGL jar files to debug and run your EGL-generated Java code. The process for providing access to those files varies by project type:

### Application client project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), do as follows:

1. For each enterprise application project that references the application client project, import jar files of interest from a directory in the file system:
  - a. In the Project Explorer view, right-click an enterprise application project and click **Import**
  - b. At the Select page, click **File System**
  - c. At the File System page, specify the directory in which the jar files reside
  - d. At the right of the page, select the jar files of interest
  - e. Click **Finish**
2. Update the manifest in the application client project so that the jar files in the enterprise application project are available at run time:
  - a. In the Project Explorer view, right-click your application client project and click **Properties**
  - b. At the left of the Properties page, click **Java JAR Dependencies**
  - c. When the section called Java JAR Dependencies is displayed at the right of the page, set each check box that corresponds to a jar file of interest
  - d. Click **OK**

### EJB project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), do as follows:

1. For each enterprise application project that references the EJB project, import jar files of interest from a directory in the file system:
  - a. In the Project Explorer view, right-click an enterprise application project and click **Import**
  - b. At the Select page, click **File System**
  - c. At the File System page, specify the directory in which the jar files reside
  - d. At the right of the page, select the jar files of interest
  - e. Click **Finish**
2. Update the manifest in the EJB project so that the jar files in the enterprise application project are available at run time:

- a. In the Project Explorer view, right-click your EJB project and click **Properties**
- b. At the left of the Properties page, click **Java JAR Dependencies**
- c. When the section called Java JAR Dependencies is displayed at the right of the page, set each check box that corresponds to a jar file of interest
- d. Click **OK**

### Java project

Before running your code with the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code with the EGL Java debugger, add entries to the project's Java build path:

1. In the Project Explorer view, right-click your Java project and click **Properties**
2. At the left of the Properties page, click **Java Build Path**
3. When the section called Java Build Path is displayed at the right of the page, click the Libraries tab
4. For each jar file to be added, click **Add External Jars** and use the Browse mechanism to select the file
5. To close the Properties page, click **OK**

### J2EE Web project

Before using the interpretive debugger, reference the non-EGL jar files in the CLASSPATH variable, as described in *Setting preferences for the EGL debugger*.

Before running your code (with or without the EGL Java debugger), import the jar files from the file system to the following Web project folder:

Web Content/WEB-INF/lib

The import process is as follows for a set of jar files in a directory:

1. In the Project Explorer view, expand the Web project, expand **Web Content**, expand **WEB-INF**, right-click **lib**, and click **Import**
2. At the Select page, click **File System**
3. At the File System page, specify the directory in which the jar files reside
4. At the right of the page, select the jar files of interest
5. Click **Finish**

The following jar-file requirements are in effect:

- A generated Java program that accesses MQSeries in any way requires MQ Series Classes for Java; in particular, the Java program needs the following jar files (although not at preparation time):
  - com.ibm.mq.jar
  - com.ibm.mqbind.jar

If you have WebSphere MQ V5.2, the software is in IBM WebSphere MQ SupportPac<sup>™</sup> MA88, which you can find by going to the IBM web site (www.ibm.com) and searching for MA88. Download and install the software; then you can access the jar files from the Java\lib subdirectory of the directory where you installed that software.

If you have WebSphere MQ V5.3, you can get the equivalent software by doing a custom install and selecting Java Messaging. Then you can access the jar files from the Java\lib subdirectory of the MQSeries installation directory.

- A generated Java program or wrapper that uses the protocol CICSJ2C to access CICS for z/OS requires access to connector.jar and cicsj2ee.jar, but only at run time. Those files are available to you when you install the CICS Transaction Gateway.

**Note:** Access of CICS is possible when the EGL Java debugger runs in J2EE. Calls to CICS are attempted but fail, however, when that debugger runs outside of J2EE or when you are using the EGL interpretive debugger, which always runs outside of J2EE.

- A generated Java program that accesses an SQL table requires a file that is installed with the database management system--

- For DB2 UDB, the file is one of the following:

```
sqllib\java\db2java.zip
sqllib\java\db2jcc.jar
```

The second of those files is available with DB2 UDB Version 8 or higher, as described in the DB2 UDB documentation.

- For Informix, the files are as follows:

```
ifxjdbc.jar
ifxjdbcx.jar
```

- For Oracle, consult the Oracle documentation.

The database file is required at run time, and can be used to validate SQL statements at preparation time.

#### **Related tasks**

“Setting preferences for the EGL debugger” on page 116

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

---

## EGL reference

---

### Assignment compatibility in EGL

Assignment-compatibility rules apply in the following situations:

- Your code assigns one non-reference variable to another; or
- EGL transfers data between an argument and the related parameter in a function invocation, but only if the parameter in the receiving function has the modifier IN (in which case the argument is the source) or OUT (in which case the parameter is the source).

Assignment compatibility is based on the following type classification:

- Text types are CHAR, MBCHAR, STRING, and UNICODE
- Numeric types are BIN, INT, BIGINT, SMALLINT, DECIMAL, NUM, NUMBER, FLOAT, SMALLFLOAT, MONEY
- Datetime types are DATE, INTERVAL, TIME, TIMESTAMP
- HEX is a type in its own category
- VisualAge Generator legacy types are DBCHAR, NUMC, and PACF, each of which follows VisualAge Generator rules

The assignment-compatibility rules are as follows:

- A field of any text type can be assigned to a field of any text type
- A field of any numeric type can be assigned to a field of any numeric type
- A field of any datetime type can be assigned to a field of any text or numeric type
- A field of type STRING or CHAR can be assigned to or from a field of type HEX
- A field of type CHAR can be assigned to or from a field of type NUM, but only if NUM has no decimals
- To assign a field of a numeric type to a field of a text type, use the system function **StrLib.formatNumber**
- To assign a field of type DATE, TIME, or TIMESTAMP to a formatted field of a text type, use the appropriate system function:
  - **StrLib.formatDate** (for dates)
  - **StrLib.formatTime** (for times)
  - **StrLib.formatTimestamp** (for timestamps)
- To assign a field of a text type to a field of type DATE, TIME, or TIMESTAMP, use the appropriate system function:
  - **ConverseLib.dateValue** (for dates)
  - **ConverseLib.timeValue** (for times)
  - **ConverseLib.timestampValue** (for timestamps)

**Note:** A different set of rules is in effect in the following cases:

- A parameter has the modifier INOUT;
- A parameter is in the onPageLoad function of a PageHandler; or
- You are assigning, passing, or returning a reference variable.

For details on those cases, see *Reference compatibility in EGL*.

The situation when passing data to a program is as follows:

- An EGL called program accepts a series of bytes without validation, except that the received data is truncated or padded in accordance with the parameter type and length
- Any errors occurs if an EGL statement attempts to use data whose format is not valid in the context of the attempted use

## Assignment across numeric types

A value of any of the numeric types (including NUMC and PACF) can be assigned to a field of any numeric type and size, and EGL does the conversions necessary to retain the value in the target format.

Non-significant zeros are added or truncated as needed. (Initial zeros in the integer part of a value are non-significant, as are trailing digits in the fraction part of a value.)

For any of the numeric types, you can use the system variable `sysVar.overflowIndicator` to test whether an assignment or arithmetic calculation resulted in an arithmetic overflow, and you can set the system variable `VGVar.handleOverflow` to specify the consequence of such an overflow.

If an arithmetic overflow occurs, the value in the target field is unchanged. If an arithmetic overflow does not occur, the value assigned to the target field is aligned in accordance with the declaration of the target field.

Let's assume that you are copying a field of type NUM to another and that the runtime value of the source field is 108.314:

- If the target field allows seven digits with one decimal place, the target field receives the value 000108.3, and a numeric overflow is *not* detected. (A loss of precision in a fractional value is not considered an overflow.)
- If the target field allows four digits with two decimal places, a numeric overflow is detected, and the value in the target field is unchanged

When you assign a floating-point value (type FLOAT or SMALLFLOAT) to a field of a fixed-point type, the target value is truncated if necessary. If a source value is 108.357 and the fixed-point target has one decimal place, for example, the target receives 108.3.

## Other cross-type assignments

Details on other cross-type assignments are as follows:

- The assignment of a value of type NUM to a target of type CHAR is valid only if the source declaration has no decimal places. This operation is equivalent to a CHAR-to-CHAR assignment.

If the source length is 4 and value is 21, for example, the content is equivalent to "0021", and a length mismatch does not cause an error condition:

- If the length of the target is 5, the value is stored as "0021 " (a single-byte space was added on the right)
- If the length of the target is 3, the value is stored as "002" (a digit was truncated on the right)

If the value of type NUM is negative and assigned to a value of type CHAR, the last byte copied into the field is an unprintable character.



- The assignment of a value of type CHAR to a target of type NUM is valid only in the following case:

- The source (a field or text expression) has digits with no other characters
- The target declaration has no decimal place

This operation is equivalent to a NUM-to-NUM assignment.

If the source length is 4 and value is "0021", for example, the content is equivalent to a numeric 21, and the effect of a length mismatch is shown in these examples:

- If the length of the target is 5, the value is stored as 00021 (a numeric zero was padded on the left)
- If the length of the target is 3, the value is stored as 021 (a non-significant digit was truncated)
- If the length of the target is 1, the value is stored as 1
- The assignment of a value of type NUMC to a target of type CHAR is possible in two steps, which eliminates the sign if the value is positive:
  1. Assign the NUMC value to a target of type NUM
  2. Assign the NUM value to a target of type CHAR

If the value of the target of type NUMC is negative, the last byte copied into the target of type CHAR is an unprintable character.
- The assignment of a value of type CHAR to a target of type HEX is valid only if the characters in the source are within the range of hexadecimal digits (0-9, A-F, a-f).
- The assignment of a value of type HEX to a target of type CHAR stores digits and uppercase letters (A-F) in the target.
- The assignment of a value of type MONEY to a target of type CHAR is not valid. The best practice for converting from MONEY to CHAR is to use the system function **strLib.formatNumber**.
- The assignment of a value of type NUM or CHAR to a target of type DATE is valid only if the source value is a valid date in accordance with the mask *yyyymmdd*; for details, see the topic *DATE*.
- The assignment of a value of type NUM or CHAR to a target of type TIME is valid only if the source value is a valid time in accordance with the mask *hhmmss*; for details, see the topic *TIME*.
- The assignment of a value of type CHAR to a target of type TIMESTAMP is valid only if the source value is a valid timestamp in accordance with the mask of the TIMESTAMP field. An example is as follows:

```
// NOT valid because February 30 is not a valid date
myTS timestamp("yyyymmdd");
myTS = "20050230";
```

If characters at the beginning of a full mask are missing (for example, if the mask is "dd"), EGL assumes that the higher-level characters ("yyyymm", in this case) represent the current moment, in accordance with the machine clock. The following statements cause a runtime error in February:

```
// NOT valid if run in February
myTS timestamp("dd");
myTS = "30";
```

- The assignment of a value of type TIME or DATE to a target of type NUM is equivalent to a NUM-to-NUM assignment.
- The assignment of a value of type TIME, DATE, or TIMESTAMP to a target of type CHAR is equivalent to a CHAR-to-CHAR assignment.



## Padding and truncation with character types

If the target is of a non-STRING character type (including DBCHAR and HEX) and has more space than is required to store a source value, EGL pads data on the right:

- Uses single-byte blanks to pad a target of type CHAR or MBCHAR
- Uses double-byte blanks to pad a target of type DBCHAR
- Uses Unicode double-byte blanks to pad a target of type UNICODE
- Uses binary zeros to pad a target of type HEX, which means (for example) that a source value "0A" is stored in a two-byte target as "0A00" rather than as "000A"

EGL truncates values on the right if the target of a character type has insufficient space to store the source value. No error is signaled.

If the target is a limited-length string, the following rules apply:

- If more characters are in the source than are valid in the target, EGL runtime truncates the copied content to fit the available length.
- If fewer characters are in the source than are valid in the target, EGL runtime pads the copied content with blanks, to the length of the target string. However, any trailing blanks in a limited-length string are ignored in a comparison.

A special case can occur in the following situation:

- The runtime platform supports the EBCDIC character set
- The assignment statement copies a literal of type MBCHAR or an item of type MBCHAR to a shorter item of type MBCHAR
- A byte-by-byte truncation would remove a final shift-in character or split a DBCHAR character

In this situation, EGL truncates characters as needed to ensure that the target item contains a valid string of type MBCHAR, then adds (if necessary) terminating single-byte blanks.

## Assignment between timestamps

If you assign an item of type TIMESTAMP to another field of type TIMESTAMP, the following rules apply:

- If the mask of the source field is missing relatively high-level entries that are required by the target field, those target entries are assigned in accordance with the clock on the machine at the time of the assignment, as shown by these examples:

```
- sourceTimeStamp timestamp ("MMdd");
  targetTimeStamp timestamp ("yyyyMMdd");

  sourceTimeStamp = "1201";

  // if this code runs in 2004, the next statement
  // assigns 20041201 to targetTimeStamp
  targetTimeStamp = sourceTimeStamp;
- sourceTimeStamp02 timestamp ("ssff");
  targetTimeStamp02 timestamp ("mmssff");

  sourceTimeStamp02 = "3201";

  // the next assignment includes the minute
  // that is current when the assignment statement runs
  targetTimeStamp02 = sourceTimeStamp02;
```

- If the mask of the source item is missing relatively low-level entries that are required by the target field, those target entries are assigned the lowest valid values, as shown by these examples:
  - sourceTimeStamp timestamp ("yyyyMM");  
targetTimeStamp timestamp ("yyyyMMdd");  
  
sourceTimeStamp = "200412";  
  
// regardless of the day, the next statement  
// assigns 20041201 to targetTimeStamp  
targetTimeStamp = sourceTimeStamp;
  - sourceTimeStamp02 timestamp ("hh");  
targetTimeStamp02 timestamp ("hhmm");  
  
sourceTimeStamp02 = "11";  
  
// regardless of the minute, the next statement  
// assigns 1100 to targetTimeStamp02  
targetTimeStamp02 = sourceTimeStamp02;

## Assignment to or from substructured fields in fixed structures

You can assign a substructured field to a non-substructured field or the reverse, and you can assign values between two substructured fields. Assume, for example, that variables named *myNum* and *myRecord* are based on the following parts:

```
DataItem myNumPart
  NUM(12)
end

Record myRecordPart type basicRecord
  10 topMost CHAR(4);
  20 next01 HEX(4);
  20 next02 HEX(4);
end
```

The assignment of a value of type HEX to an item of type NUM is not valid outside of the mathematical system variables; but an assignment of the form **myNum = topMost** is valid because **topMost** is of type CHAR. In general terms, the primitive types of the fields in the assignment statement guide the assignment, and the primitive types of subordinate items are not taken into account.

The primitive type of a substructured item is CHAR by default. If you assign data to or from a substructured field and do not specify a different primitive type at declaration time, the rules described earlier for fields of type CHAR are in effect during the assignment.

## Assignment of a fixed record

An assignment of one fixed record to another is equivalent to assigning one substructured item of type CHAR to another. A mismatch in length adds single-byte blanks to the right of the received value or removes single-byte characters from the right of the received value. The assignment does not consider the primitive types of subordinate structure fields.

The following exceptions apply:

- The content of a record can be assigned to a record or to a field of type CHAR, HEX, or MBCHAR, but not to a field of any other type
- A record can receive data from a record or from a string literal or from a field of type CHAR, HEX, or MBCHAR, but not from a numeric literal or from a field of a type other than CHAR, HEX, or MBCHAR

Finally, if you assign an SQL record to or from a record of a different type, you must ensure that the non-SQL record has space for the four-byte area that precedes each structure field.

#### Related concepts

“PageHandler” on page 223

#### Related reference

“Assignments”

“DATE” on page 41

“EGL statements” on page 88

“formatNumber()” on page 1007

“Function parameters” on page 616

“Function part in EGL source format” on page 621

“handleOverflow” on page 1083

“move” on page 716

“overflowIndicator” on page 1069

“PageHandler part in EGL source format” on page 785

“Primitive types” on page 34

“Program parameters” on page 840

“Program part in EGL source format” on page 841 “Substrings” on page 882

“TIME” on page 44

---

## Assignments

An EGL assignment copies data from one area of memory to another and can copy the result of a numeric or text expression into a source field.

►► `target = source ;` ◄◄

**target** A field, record, fixed record, or system variable.

You can specify a substring on the left side of an assignment statement if the target field is of type CHAR, DBCHAR, or UNICODE. The substring area is filled (padded with blanks, if necessary), and the assigned text does not extend beyond the substring area but is truncated, if necessary. For syntax details, see *Substrings*.

**source** A record, fixed record, or a numeric or character expression

Examples of assignments are as follows:

```
z = a + b + c;  
myDate = VVar.currentShortGregorianDate;  
myUser = sysVar.userID;  
myRecord01 = myRecord02;  
myRecord02 = "USER";
```

The behavior of an EGL assignment statement is different from that of a **move** statement, which is described in *move*.

The assignment rules are described in *Assignment compatibility in EGL*.

#### Related concepts

“Syntax diagram for EGL statements and commands” on page 884

#### Related reference

"Assignment compatibility in EGL" on page 451

"move" on page 716

"Substrings" on page 882

---

## Association elements

As described in *Resource associations*, the resource associations part is composed of association elements. Each element is specific to a file name (property "fileName" on page 458) and contains a set of entries, each with these properties:

- "system" on page 458
- "fileType"

The values of the **system** and **fileType** properties determine what additional properties are available to you from the following list:

- "commit"
- "conversionTable"
- "formFeedOnClose" on page 458
- "replace" on page 458
- "systemName" on page 459
- "text" on page 459

### commit

Indicates (for an EGL-generated Java program on iSeries) whether to enable commitment control.

Select one of these values:

#### NO (the default)

Use of sysLib.commit or sysLib.rollback has no effect.

#### YES

You can use sysLib.commit and sysLib.rollback to define the end of a logical unit of work.

### conversionTable

Specifies the name of the conversion table used by a generated Java program during access of an MQSeries message queue.

For additional information, see *Data conversion*.

### fileType

Specifies the file organization on the target system. You can select an explicit type like *seqws*. Alternatively, you can select the value *default*, which is itself the default value of the property **fileType**. Use of the default means that a file type is selected automatically:

- For a particular combination of target system and EGL record type; or
- For print output, when the file name is *printer*.

*Record and file type cross-reference* shows the explicit **fileType** values, as well as the value used if you select *default*.

## fileName

Refers to a logical file name, as specified in one or more records. You are creating an association element that relates this name to a physical resource on one or more target systems. (For print output, specify the value *printer*.)

You can use an asterisk (\*) as a global substitution character in a logical file name; however, that character is valid only as the last character. For details, see *Resource associations and file types*.

## formFeedOnClose

Indicates whether a form feed is issued when the output of a print form ends. (A print form is produced when your code issues a **print** statement.)

This property is available only if the **fileName** value is *printer* in one of the following cases:

- The **system** value is *aix*, *iSeriesj*, or *linux*, and the **fileType** value is *seqws* or *spool*; or
- The **system** value is *win*, and the **fileType** value is *seqws*.

Select one of these values:

**YES**

A form feed occurs (the default)

**NO**

A form feed does not occur

## replace

Specifies whether adding a record to the file replaces the file rather than appending to the file. This entry is used only in these cases:

- You are generating Java code; and
- The record is of file type **seqws**.

Select one of these values:

**NO**

Append to the file (the default)

**YES**

Replace the file

## system

Specifies the target platform. Select one of the following values:

**aix**

AIX

**imsbmp**

IMS BMP

**imsvs**

IMS/VS

**iseriesj**

iSeries

**linux**

Linux

**win**

Windows 2000/NT/XP

**any**

Any target platform; for details, see *Resource associations and file types*

## **systemName**

Specifies the system resource name of the file or data set associated with the file name. Enclose the value in single or double quote marks if a space or any of the following characters is in the value:

% = , ( ) /

## **text**

Specifies whether to cause a generated Java program to do the following when accessing a file by way of a serial record:

- Append end-of-line characters during the **add** operation. On non-UNIX platforms, those characters are a carriage return and linefeed; on UNIX platforms, the only character is a linefeed.
- Remove end-of-line characters during the **get** or **get next** operation.

Select one of these values:

**NO**

The default is not to append or remove the end-of-line characters

**YES**

Make the changes, as is useful if the generated program is exchanging data with products that expect records to end with the end-of-line characters

### **Related concepts**

"Resource associations and file types" on page 393

### **Related task**

"Adding a resource associations part to an EGL build file" on page 397

"Editing a resource associations part in an EGL build file" on page 397

"Removing a resource associations part from an EGL build file" on page 398

### **Related reference**

"Data conversion" on page 558

"I/O error values" on page 638

"Record and file type cross-reference" on page 860

---

## **asynchLink element**

An *asynchLink* element of a linkage options part specifies how a generated program invokes another program asynchronously, as occurs when the originating program invokes the system function `sysLib.startTransaction`.

You can avoid specifying an *asynchLink* element if you accept the default behavior, which assumes that the created transaction is to be started from the same Java package.

Each element includes the property `recordName`, which references a record that is also referenced in the specific `sysLib.startTransaction` function whose action is being modified.

The other property is **package**, which is needed only if the source for the invoked program is in a package that is different from the invoker's package.

#### Related concepts

"Linkage options part" on page 399

#### Related reference

"package in `asynchLink` element" on page 461

"recordName in `asynchLink` element" on page 461

## csouidpwd.properties file for remote calls

In a situation described later, you must create and provide access to the file **csouidpwd.properties**. That file includes authentication details needed for a remote call from a Java program or wrapper.

The situation is as follows:

- The linkage options part, `callLink` element, property **remoteComType** is set to `JAVA400`, `CICSJ2C`, or `CICSECI`; and
- A user ID and password are required; and
- One of these cases applies:
  - The call is made from a Java program, but the code does not first invoke the system function `SysLib.setRemoteUser` with values other than blanks; or
  - The call is made from a Java wrapper, but the Java code that includes that wrapper has not invoked the `CSOCallOptions` methods `setUserId` and `setPassword` with values other than blanks.

If the invocation of `SysLib.setRemoteUser` (or the invocation of the appropriate `CSOCallOptions` method) provides a blank user ID or password, the value of the equivalent property is sought in **csouidpwd.properties**.

Your task is as follows:

1. Create the file **csouidpwd.properties**, which can contain property settings that are formatted as follows, each on a separate line:

**CSOUID=userid**

*userid* is the user ID for the remote call

**CSOPWD=password**

*password* is the password for the remote call

2. Ensure that the file is a directory that is referenced by the classpath. An appropriate directory is your project's `JavaSource` folder.

#### Related concepts

"Java wrapper" on page 390

#### Related reference

"Java wrapper classes" on page 652  
"remoteComType in `callLink` element" on page 511  
"setRemoteUser()" on page 1039

## package in asynchLink element

The linkage options part, asynchLink element, property **package** specifies the name of the package that contains the program being invoked. The default is the package of the invoking program.

The package name that is used in generated Java programs is the package name of the EGL program, but in lower case; and when EGL generates output from the asynchLink element, the value of **package** is changed (if necessary) to lower case.

### Related concepts

“Linkage options part” on page 399

### Related reference

“asynchLink element” on page 459

“recordName in asynchLink element”

## recordName in asynchLink element

The linkage options part, asynchLink element, property **recordName** specifies the name of the record that is used in the system function sysLib.startTransaction. In this case, the record name is used to identify which program or transaction is associated with the asynchLink element.

You can use an asterisk (\*) as a global substitution character in the record name; however, that character is valid only as the last character. For details, see *Linkage options part*.

### Related concepts

“Linkage options part” on page 399

### Related reference

“asynchLink element” on page 459

“package in asynchLink element”

“startTransaction()” on page 1041

---

## Basic record part in EGL source format

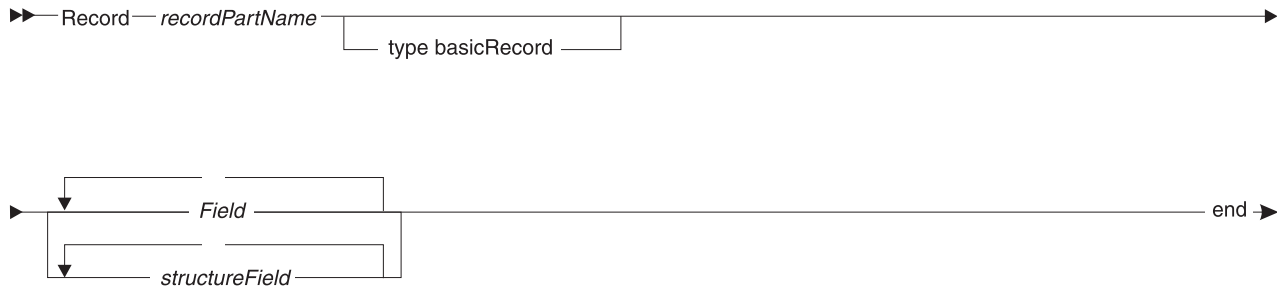
You declare a record part of type basicRecord in an EGL source file, which is described in *EGL source format*.

An example of a basic record part is as follows:

```
Record myBasicRecordPart type basicRecord
  10 myField01 CHAR(2);
  10 myField02 CHAR(78);
end
```

The syntax diagram for a basic record is as follows:





### **Record** *recordPartName* **basicRecord**

Identifies the part as being of type `basicRecord` and specifies the name. For rules, see *Naming conventions*.

### *field*

A variable appropriate in a record, as described in *Record parts*. End each variable declaration with a semicolon.

### *structureField*

A fixed-structure field, as described in *Structure field in EGL source format*.

### **Related concepts**

- "EGL projects, packages, and files" on page 15
- "Fixed record parts" on page 136
- "References to parts" on page 23
- "Parts" on page 19
- "Record parts" on page 135
- "References to variables in EGL" on page 59
- "Typedef" on page 28

### **Related tasks**

- "Syntax diagram for EGL statements and commands" on page 884

### **Related reference**

- "DataItem part in EGL source format" on page 566
- "EGL source format" on page 586
- "Function part in EGL source format" on page 621
- "Indexed record part in EGL source format" on page 632
- "MQ record part in EGL source format" on page 769
- "Naming conventions" on page 778
- "Primitive types" on page 34
- "Program part in EGL source format" on page 841
- "Properties that support variable-length records" on page 860
- "Relative record part in EGL source format" on page 865
- "Serial record part in EGL source format" on page 868
- "SQL record part in EGL source format" on page 877
- "Structure field in EGL source format" on page 880

## **Build parts**

### **EGL build-file format**

The structure of a `.eglbld` file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGL PUBLIC "-//IBM//DTD EGL 5.1//EN" "">
<EGL>
  <!-- place your import statements here -->
  <!-- place your parts here -->
</EGL>
```

Your task is to place import statements and parts inside the <EGL> element.

You specify <import> elements to reference the file containing the next build descriptor in a chain or to reference any of the build parts referenced by a build descriptor. An example of an import statement is as follows:

```
<import file="myBldFile.egl.bld"/>
```

You declare parts from this list:

- <BuildDescriptor>
- <LinkageOptions>
- <ResourceAssociations>

A simple example is as follows:

```
<EGL>
  <import file="myBldFile.egl.bld"/>
  <BuildDescriptor name="myBuildDescriptor"
    genProject="myNextProject"
    system="WIN"
    J2EE="NO"
    genProperties="GLOBAL"
    genDataTables="YES"
    dbms="DB2"
    sqlValidationConnectionURL="jdbc:db2:SAMPLE"
    sqlJDBCClass="COM.ibm.db2.jdbc.app.DB2Driver"
    sqlDB="jdbc:db2:SAMPLE"
  </BuildDescriptor>
</EGL>
```

You can review the build-file DTD, which is in the following subdirectory:

```
installationDir\egl\ eclipse\plugins\
com.ibm.etools.egl_version\ dtd
```

*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The installed version of the plugin; for example, 6.0.0

The file name (like *egl\_wssd\_6\_0.dtd*) begins with the letters *egl* and an underscore. The characters *wssd* refer to Rational Web Developer and Rational Application Developer; the characters *wsed* refer to Rational Application Developer for z/OS; and the characters *wdsc* refer to Rational Application Developer for iSeries.

### Related concepts

“Import” on page 33

“Parts” on page 19

#### Related tasks

“Creating an EGL source file” on page 130

#### Related reference

“EGL editor” on page 577

## Build descriptor options

The next table lists all the build descriptor options.

Build descriptor option	Build option filter(s)	Description
<b>buildPlan</b>	<ul style="list-style-type: none"><li>• Java target</li></ul>	Specifies whether a build plan is created
<b>cicsj2cTimeout</b>	<ul style="list-style-type: none"><li>• Debug</li><li>• Java target</li><li>• Java iSeries</li></ul>	Assigns a value to the Java runtime property <b>cso.cicsj2c.timeout</b> , which specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C
<b>commentLevel</b>	<ul style="list-style-type: none"><li>• Java target</li><li>• Java iSeries</li></ul>	Specifies the extent to which EGL system comments are included in output source code
<b>currencySymbol</b>	<ul style="list-style-type: none"><li>• Debug</li><li>• Java target</li></ul>	Specifies a currency symbol that is composed of one to three characters
<b>dbContentSeparator</b>	<ul style="list-style-type: none"><li>• Debug</li><li>• Java target</li><li>• Java iSeries</li></ul>	Specifies the type of database accessed by the generated program
<b>dbms</b>	<ul style="list-style-type: none"><li>• Debug</li><li>• Java target</li><li>• Java iSeries</li></ul>	Specifies the type of database accessed by the generated program
<b>decimalSymbol</b>	<ul style="list-style-type: none"><li>• Debug</li><li>• Java target</li><li>• Java iSeries</li></ul>	Assigns a character to the Java runtime property <b>vgj.nls.number.decimal</b> , which indicates what character is used as a decimal symbol
<b>defaultDateFormat</b>	<ul style="list-style-type: none"><li>• Debug</li><li>• Java target</li><li>• Java iSeries</li></ul>	Specifies the generated value for the Java runtime property <b>vgj.default.dateFormat</b> . That property sets the initial runtime value of system variable <b>StrLib.defaultDateFormat</b> , which contains one of the masks that can be used to create the string returned by the function <b>StrLib.formatDate</b>

Build descriptor option	Build option filter(s)	Description
<b>defaultMoneyFormat</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	<p>Specifies the generated value for the Java runtime property <b>vgj.default.moneyFormat</b>. That property sets the initial runtime value of system variable <b>StrLib.defaultMoneyFormat</b>, which contains one of the masks that can be used to create the string returned by the function <b>StrLib.formatNumber</b></p>
<b>defaultNumericFormat</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	<p>Specifies the generated value for the Java runtime property <b>vgj.default.numericFormat</b>. That property sets the initial runtime value of system variable <b>StrLib.defaultNumericFormat</b>, which contains one of the masks that can be used to create the string returned by the function <b>StrLib.formatNumber</b></p>
<b>defaultTimeFormat</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	<p>Specifies the generated value for the Java runtime property <b>vgj.default.timeFormat</b>. That property sets the initial runtime value of system variable <b>StrLib.defaultTimeFormat</b>, which contains one of the masks that can be used to create the string returned by the function <b>StrLib.formatNumber</b></p>
<b>defaultTimeStampFormat</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	<p>Specifies the generated value for the Java runtime property <b>vgj.default.timestampFormat</b>. That property sets the initial runtime value of system variable <b>StrLib.defaultTimeStampFormat</b>, which contains one of the masks that can be used to create the string returned by the function <b>StrLib.formatTimeStamp</b></p>
<b>destDirectory</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	<p>Specifies the name of the directory that stores the output of preparation, but only when you generate Java</p>
<b>destHost</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	<p>Specifies the name or numeric TCP/IP address of the target machine where the build server resides</p>

Build descriptor option	Build option filter(s)	Description
<b>destPassword</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	Specifies the password that EGL uses to log on to the machine where preparation occurs
<b>destPort</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	Specifies the port on which a remote build server is listening for build requests
<b>destUserID</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	Specifies the user ID that EGL uses to log on to the machine where preparation occurs
<b>eliminateSystemDependentCode</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Indicates whether, at validation time, EGL ignores code that will never run in the target system.
<b>enableJavaWrapperGen</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies whether to allow generation of Java wrapper classes
<b>genDataTables</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Indicates whether you want to generate data tables
<b>genDirectory</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	Specifies the fully qualified path of the directory into which EGL places generated output and preparation-status files
<b>genFormGroup</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	Indicates whether you want to generate the form group that is referenced in the use declaration of the program you are generating
<b>genHelpFormGroup</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	Indicates whether you want to generate the help form group that is referenced in the use declaration of the program you are generating.
<b>genProject</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Places the output of Java generation into a workbench project and automates tasks that are required for Java runtime setup
<b>genProperties</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies what kind of Java runtime properties to generate (if any) and, in some cases, whether to generate a linkage properties file
<b>genResourceBundle</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies what kind of Java runtime properties to generate (if any) and, in some cases, whether to generate a linkage properties file

Build descriptor option	Build option filter(s)	Description
<b>genVGUIRecords</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Indicates whether you want to generate the VGUI records that are referenced by a program of type VGWebTransaction
<b>itemsNullable</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies the circumstance in which your code can set primitive fields to NULL
<b>J2EE</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies whether a Java program is generated to run in a J2EE environment
<b>J2EELevel</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies the J2EE level of the Web application server to which an EGL-generated Web service or service-binding library will be deployed
<b>linkage</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Contains the name of the linkage options part that guides aspects of generation
<b>nextBuildDescriptor</b> (see Build descriptor part)	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Identifies the next build descriptor in chain
<b>prep</b>	<ul style="list-style-type: none"> <li>• Java target</li> </ul>	Specifies whether EGL begins preparation when generation completes with a return code $\leq 4$
<b>resourceAssociations</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Contains the name of a resource associations part, which relates record parts to files and queues on the target platforms
<b>resourceBundleLocale</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies a Java locale to be included in the name of a resource bundle that is generated for a VGUI record or for a message table, either of which is used in a VGWebTransaction program
<b>serverType</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies the type of Web application server to which an EGL-generated Web service or service-binding library will be deployed
<b>serviceRuntime</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies the runtime code that will be used to handle an EGL service, an EGL-generated Web service, or a service-binding library that references a Web service (whether or not generated by EGL)

Build descriptor option	Build option filter(s)	Description
<b>sessionBeanID</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Identifies the name of an existing session element in the J2EE deployment descriptor
<b>sqlCommitControl</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Allows generation of a Java runtime property that specifies whether a commit occurs after every change to the default database
<b>sqlIDB</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies the default database used by a generated program
<b>sqlID</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies a user ID that is used to connect to a database during generation-time validation of SQL statements or at run time
<b>sqlJDBCClass</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies a driver class that is used to connect to a database during generation-time validation of SQL statements or during a non-J2EE Java debugging session
<b>sqlJNDIName</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies the default database used by a generated Java program that runs in J2EE
<b>sqlPassword</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies a password that is used to connect to a database during generation-time validation of SQL statements or at run time
<b>sqlValidationConnectionURL</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies a URL that is used to connect to a database during generation-time validation of SQL statements
<b>synchOnTrxTransfer</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Indicates how the EGL runtime handles resource commitment in some cases, when the transfer statement runs in a main text or basic program generated for Java or written for IMS BMP or z/OS batch
<b>system</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies a category of generation output
<b>targetNLS</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies the target national language code used for runtime output

Build descriptor option	Build option filter(s)	Description
<b>tempDirectory</b>	<ul style="list-style-type: none"> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Identifies the directory in which to store the JSP files when you are generating a VGWebTransaction program or VGUIRecord into a Web project
<b>userMessageFile</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Specifies the generated value for Java runtime property <b>vgj.messages.file</b> , which specifies a properties file that includes messages you create or customize
<b>VAGCompatibility</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Indicates whether the generation process allows use of special program syntax
<b>validateSQLStatements</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	Indicates whether SQL statements are validated against a database
<b>webServiceEncodingStyle</b>	<ul style="list-style-type: none"> <li>• Debug</li> <li>• Java target</li> <li>• Java iSeries</li> </ul>	For service generation, determines the format of the SOAP messages sent to and from an EGL-generated Web service

#### Related concepts

“Build descriptor part” on page 383

“Java runtime properties” on page 431

#### Related tasks

“Adding a build descriptor part to an EGL build file” on page 387

“Editing general options in a build descriptor” on page 388

#### Related reference

“Java runtime properties (details)” on page 642

### associatedJavaPartBuildDescriptor

The build descriptor option **associatedJavaPartBuildDescriptor** specifies

#### Related reference

“Build descriptor options” on page 464

### buildPlan

The build descriptor option **buildPlan** specifies whether a build plan is created. Valid values are YES and NO, and the default is YES.

The build plan is placed in the directory identified by build descriptor option **genDirectory**.

A special case is in effect when you generate Java code into a project. Then, no build plan is created regardless of the setting of **buildPlan**, but preparation occurs in either of two situations:

- Whenever you rebuild the project



- Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

You may wish to create a build plan and to invoke that plan at a later time. For details, see *Invoking a build plan after generation*.

#### Related concepts

“Build plan” on page 413

#### Related tasks

“Invoking a build plan after generation” on page 419

#### Related reference

“Build descriptor options” on page 464

### cicsj2cTimeout

When you are generating Java code, the build descriptor option **cicsj2cTimeout** assigns a value to the Java runtime property **cso.cicsj2c.timeout**. That property specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C.

The default value of the runtime property is 30000, which represents 30 seconds. If the value is set to 0, no timeout occurs. The value must be greater than or equal to 0.

The property **cso.cicsj2c.timeout** has no effect on calls when the called program is running in WebSphere 390; for details, see *Setting up the J2EE server for CICSJ2C calls*.

#### Related concepts

“Java runtime properties” on page 431

#### Related tasks

“Setting up the J2EE server for CICSJ2C calls” on page 441

#### Related reference

“Build descriptor options” on page 464

“Java runtime properties (details)” on page 642

### commentLevel

The build descriptor option **commentLevel** specifies the extent to which EGL system comments are included in output source code.

Valid values are as follows:

- 0** Minimal comments are in the output, which includes comments on any name aliases that EGL generates
- 1** In addition to the comments included with level 0, scripting statements are placed immediately before the code that is generated to implement those statements.

The default is 1.

Raising the comment level has no effect on the size or performance of the prepared code, but increases the size of the output and the time needed to generate, transfer, and prepare the output.

**Related reference**

"Build descriptor options" on page 464

**currencySymbol**

The build descriptor option **currencySymbol** is available only for Java output and specifies a currency symbol that is composed of one to three characters. If you do not specify this option, the default value is derived from the locale of the system on which you generate output.

To specify a character that is not on your keyboard, hold down the **Alt** key and use the numeric key pad to type the character's decimal code. The decimal code for the Euro, for example, is 0128 on Windows 2000/NT/XP.

**Related concepts**

"Build descriptor part" on page 383

**Related reference**

"Build descriptor options" on page 464

**dbContentSeparator**

The build descriptor option **dbContentSeparator** specifies the symbol used to separate one value from the next in the system functions **SysLib.loadTable** and **SysLib.unLoadTable**.

The value is copied to the Java runtime property **vgj.default.databaseDelimiter**. If no value is specified in that property at run time, the default value is a pipe (|).

**Related concepts**

"Java runtime properties" on page 431

**Related reference**

"Build descriptor options" on page 464

"Java runtime properties (details)" on page 642

"loadTable()" on page 1033

"unloadTable()" on page 1043

**dbms**

The build descriptor option **dbms** specifies the type of database accessed by the generated program. Select one of the following values:

- DB2 (the default value)
- INFORMIX
- ORACLE

**Related reference**

"Build descriptor options" on page 464

"Informix and EGL" on page 299

**decimalSymbol**

When you are generating Java code, the build descriptor option **decimalSymbol** assigns a character to the Java runtime property **vgj.nls.number.decimal**, which indicates what character is used as a decimal symbol. If you do not specify the build descriptor option **decimalSymbol**, the character is determined by the locale associated with the Java runtime property **vgj.nls.code**.

The value can be no more than one character.

#### Related concepts

“Java runtime properties” on page 431

#### Related reference

“Build descriptor options” on page 464

“Java runtime properties (details)” on page 642

### defaultDateFormat (EGL build descriptor option)

The build descriptor option **defaultDateFormat** specifies the generated value for the Java runtime property **vgj.default.dateFormat**. That property sets the initial runtime value of system variable **StrLib.defaultDateFormat**, which contains one of the masks that can be used to create the string returned by the function **StrLib.formatDate**.

For details on valid values, see *formatNumber()*.

#### Related concepts

“Java runtime properties” on page 431

#### Related reference

“Build descriptor options” on page 464

“DATE” on page 41

“Date, time, and timestamp format specifiers” on page 46

“defaultDateFormat (EGL system variable)” on page 1004

“formatDate()” on page 1007

“Java runtime properties (details)” on page 642

### defaultMoneyFormat (EGL build descriptor option)

The build descriptor option **defaultMoneyFormat** specifies the generated value for the Java runtime property **vgj.default.moneyFormat**. That property sets the initial runtime value of system variable **StrLib.defaultMoneyFormat**, which contains one of the masks that can be used to create the string returned by the function **StrLib.formatNumber**.

For details on valid values, see *formatNumber()*.

#### Related concepts

“Java runtime properties” on page 431

#### Related reference

“Build descriptor options” on page 464

“defaultMoneyFormat (EGL system variable)” on page 1004

“formatNumber()” on page 1007

“Java runtime properties (details)” on page 642

### defaultNumericFormat (EGL build descriptor option)

The build descriptor option **defaultNumericFormat** specifies the generated value for the Java runtime property **vgj.default.numericFormat**. That property sets the initial runtime value of system variable **StrLib.defaultNumericFormat**, which contains one of the masks that can be used to create the string returned by the function **StrLib.formatNumber**.

For details on valid values, see *formatNumber()*.

#### Related concepts

“Java runtime properties” on page 431

#### Related reference

"Build descriptor options" on page 464  
"defaultNumericFormat (EGL system variable)" on page 1005  
"formatNumber()" on page 1007  
"Java runtime properties (details)" on page 642

#### defaultTimeFormat (EGL build descriptor option)

The build descriptor option **defaultTimeFormat** specifies the generated value for the Java runtime property **vgj.default.timeFormat**. That property sets the initial runtime value of system variable **StrLib.defaultTimeFormat**, which contains one of the masks that can be used to create the string returned by the function **StrLib.formatTime**.

For details on valid values, see *Date, time, and timestamp specifiers*.

#### Related concepts

"Java runtime properties" on page 431

#### Related reference

"Build descriptor options" on page 464  
"Date, time, and timestamp format specifiers" on page 46  
"defaultTimeFormat (system variable)" on page 1005  
"formatTime()" on page 1008  
"Java runtime properties (details)" on page 642  
"TIME" on page 44

#### defaultTimeStampFormat (EGL build descriptor option)

The build descriptor option **defaultTimeStampFormat** specifies the generated value for the Java runtime property **vgj.default.timestampFormat**. That property sets the initial runtime value of system variable **StrLib.defaultTimeStampFormat**, which contains one of the masks that can be used to create the string returned by the function **StrLib.formatTimeStamp**.

For details on valid values, see *Date, time, and timestamp specifiers*.

#### Related concepts

"Java runtime properties" on page 431

#### Related reference

"Build descriptor options" on page 464  
"Date, time, and timestamp format specifiers" on page 46  
"defaultTimeStampFormat (EGL system variable)" on page 1005  
"formatTimeStamp()" on page 1009  
"Java runtime properties (details)" on page 642  
"TIMESTAMP" on page 44

#### destDirectory

The build descriptor option **destDirectory** specifies the directory that stores the output of preparation. This option is meaningful only when you generate into a directory rather than into a project.

When you specify a fully qualified file path, all but the last directory must exist. If you specify `c:\buildout` on Windows 2000, for example, EGL creates the buildout directory if it does not exist. If you specify `c:\interim\buildout` and the interim directory does not exist, however, preparation fails.

If you specify a relative directory (such as myid/mysource on USS), the output is placed in the bottom-most directory, which is relative to the default directory, as described next.

The default value of **destDirectory** is affected by the status of build descriptor option **destHost**:

- If **destHost** is specified, the default value of **destDirectory** is the directory in which the build server was started
- If **destHost** is not specified, preparation occurs on the machine where generation occurs, and the default value of **destDirectory** is given by build descriptor option **genDirectory**

The user specified by build descriptor option **destUserID** must have the authority to write to the directory that receives the output of preparation.

You cannot use a UNIX variable (\$HOME, for example) to identify part of a directory structure on USS.

#### Related reference

“Build descriptor options” on page 464

“destHost”

“genProject” on page 478

#### destHost

The build descriptor option **destHost** specifies the name or numeric TCP/IP address of the target machine where the build server resides. No default is available.

If you are preparing Java output, the following statements apply:

- **destHost** is optional
- **destHost** is meaningful only if you generate into a directory rather than into a project
- If you specify **destHost** without specifying **destDirectory**, the directory in which the build server was started is the one that receives source and preparation outputs
- If you do not specify **destHost**, preparation occurs on the machine where generation occurs; and if **destDirectory** is not specified, the directory that is specified by build descriptor option **genDirectory** is the one that receives source and preparation outputs
- The UNIX environments are case sensitive

You can type up to 64 characters for the name or TCP/IP address. If you are developing on Windows NT®, you must specify a name rather than a TCP/IP address.

Two example values for **destHost** are as follows:

abc.def.ghi.com

9.99.999.99

#### Related reference

“Build descriptor options” on page 464

"destDirectory" on page 473

"destPassword"

"destPort"

### **destPassword**

The build descriptor option **destPassword** specifies the password that EGL uses to log on to the machine where preparation occurs.

This option and the description on this page are meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

The password provides access for the userid specified in build descriptor option **destUserID**. The value of the password is case sensitive for all target systems.

No default is available.

Use of **destPassword** means that a password is stored in an EGL build file. You can avoid the security risk by not setting the build descriptor option. When you start generation, you can set the password in an interactive generation dialog or on the command line.

#### **Related reference**

"Build descriptor options" on page 464

"destHost" on page 474

"destUserID"

### **destPort**

The build descriptor option **destPort** specifies the port on which a remote build server is listening for build requests.

This option is meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

No default value is available.

#### **Related reference**

"Build descriptor options" on page 464

"destHost" on page 474

### **destUserID**

The build descriptor option **destUserID** specifies the userid that EGL uses to log on to the machine where preparation occurs.

This option and the description on this page are meaningful only if you are generating into a directory rather than into a project and only if you specify a value for build descriptor option **destHost**.

The user specified by **destUserID** must have the authority to write to the directory. The option value is case sensitive for all target systems.

No default is available.

### Related reference

“Build descriptor options” on page 464

“destHost” on page 474

“destPassword” on page 475

## eliminateSystemDependentCode

The build descriptor option **eliminateSystemDependentCode** indicates whether, at validation time, EGL ignores code that will never run in the target system. Valid values are *yes* (the default) and *no*. Specify *no* only if the output of the current generation will run in multiple systems.

The option **eliminateSystemDependentCode** is meaningful only in relation to the system function **sysVar.systemType**. That function does not itself affect what code is validated at generation time. For example, the following **add** statement may be validated even if you are generating for Windows:

```
if (sysVar.systemType IS AIX)
  add myRecord;
end
```

To avoid validating code that will never run in the target system, take either of the following actions:

- Set the build descriptor option **eliminateSystemDependentCode** to *yes*. In the current example, the **add** statement is not validated if you set that build descriptor option to *yes*. Be aware, however, that the generator can eliminate system-dependent code only if the logical expression (in this case, `sysVar.systemType IS AIX`) is simple enough to evaluate at generation time.
- Alternatively, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

```
if (sysVar.systemType IS AIX)
  call myAddProgram myRecord;
end
```

### Related concepts

“Build descriptor part” on page 383

### Related reference

“Build descriptor options” on page 464

## enableJavaWrapperGen

When you issue the commands to generate a program, the build descriptor option **enableJavaWrapperGen** allows you to choose from three alternatives:

### YES (the default)

Generate the program and allow generation of the related Java wrapper classes and (if appropriate) the related EJB session bean

### ONLY

Do not generate the program, but allow generation of the related Java wrapper classes and (if appropriate) the related EJB session bean

### NO

Generate the program, but not the Java wrapper classes or the related EJB session bean, if any

Actual generation of the Java wrapper classes and EJB session bean requires appropriate settings in the linkage options part that is used at generation time. For an overview, see *Java wrapper*.

### Related concepts

"Java wrapper" on page 390

### Related reference

"Java wrapper classes" on page 652

## genDataTables

The build descriptor option **genDataTables** indicates whether you want to generate the data tables that are referenced in the program you are generating. The references are in the program's use declaration and in the program property **msgTablePrefix**.

Valid values are *yes* (the default) and *no*.

Set the value to *no* in the following case:

- The data tables referenced in the program were previously generated; and
- Those tables have not changed since they were last generated.

For other details, see *DataTable*.

### Related concepts

"Build descriptor part" on page 383

"DataTable" on page 176

### Related reference

"Build descriptor options" on page 464

"Program part in EGL source format" on page 841

"Use declaration" on page 1091

## genDirectory

The build descriptor option **genDirectory** specifies the fully qualified path of the directory into which EGL places generated output and preparation-status files.

When you are generating in the workbench or from the workbench batch interface, the following rules apply:

### For Java generation

If you are generating Java code for iSeries, you are required to specify **genProject** rather than **genDirectory**. Otherwise, you can specify either **genProject** or **genDirectory**. If both are specified, an error results. If neither is specified, Java output is generated into the project that contains the EGL source file being generated.

If you are generating from the EGL SDK, the following rules apply:

- You must specify **genDirectory**
- An error results if you specify **genProject**
- You cannot generate Java code for iSeries

For details on deploying Java code, see *Processing Java code that is generated into a directory*.

### Related concepts

"Generation from the EGL Software Development Kit (SDK)" on page 418

"Generation from the workbench batch interface" on page 417

"Generation in the workbench" on page 416



### Related tasks

“Processing Java code that is generated into a directory” on page 420

### Related reference

“Build descriptor options” on page 464

“genDirectory” on page 477

“genProject”

## genFormGroup

The build descriptor option **genFormGroup** indicates whether you want to generate the form group that is referenced in the use declaration of the program you are generating. Valid values are *yes* (the default) and *no*.

The help form group, if any, is not affected by this option, but by the build descriptor option **genHelpFormGroup**.

### Related concepts

“Build descriptor part” on page 383

### Related reference

“Build descriptor options” on page 464

“genHelpFormGroup”

“Use declaration” on page 1091

## genHelpFormGroup

The build descriptor option **genHelpFormGroup** indicates whether you want to generate the help form group that is referenced in the use declaration of the program you are generating. Valid values are *yes* (the default) and *no*.

The main form group is not affected by this option, but by the build descriptor option **genFormGroup**.

### Related concepts

“Build descriptor part” on page 383

### Related reference

“Build descriptor options” on page 464

“genFormGroup”

“Use declaration” on page 1091

## genProject

The build descriptor option **genProject** places the output of Java generation into a Workbench project and automates tasks that are required for Java runtime setup. For details on that setup and on the benefits of using **genProject**, see *Generation of Java code into a project*.

To use **genProject**, specify the project name. EGL then ignores the build descriptor options **buildPlan**, **genDirectory**, and **prep**, and preparation occurs in either of two cases:

- Whenever you rebuild the project
- Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

If you set the option **genProject** to the name of a project that does not exist in the workbench, EGL uses the name to create a Java project, except in these cases:

- If you are generating a PageHandler and specify a project different from the one that contains the related JSP and if that other project does not exist, EGL creates an EGL Web project. (However, it is recommended that you generate the PageHandler into the project that contains the related JSP.)
- A second exception concerns EJB processing and occurs if you are generating a Java wrapper when the linkage options part, callLink element, **type** property is ejbCall (for the call from the wrapper to the EGL-generated program). In that case, EGL uses the value of **genProject** to create an EJB project and creates a new enterprise application project (if necessary) with a name that is the same as the EJB project name plus the letters EAR.

In addition to creating a project, EGL does as follows:

- EGL creates folders in the project. The package structure begins under the top-level JavaSource folder, which is within the project's Java Resources folder. You may change the name JavaSource by right-clicking on the folder name and selecting **Refactor**.
- If a JRE definition is specified in the preferences page for Java (installed JREs), EGL adds the classpath variable JRE\_LIB. That variable contains the path to the runtime JAR files for the JRE currently in use.

When you are generating in the Workbench or from the Workbench batch interface, the following rules apply:

#### For Java generation

If you are generating Java code for iSeries, you are required to specify **genProject**. Otherwise, you are not required to specify either **genProject** or **genDirectory**. If neither is specified, Java output is generated into the project that contains the EGL source file being generated.

If you are generating a PageHandler, and the project specified exists, then the project must be an EGL Web project. If you are generating a session EJB, and the project specified exists, then the project must be an EJB project.

#### For COBOL generation

You must specify **genDirectory**, and in most cases EGL ignores any setting for **genProject**. If the COBOL program of type VGWebTransaction presents a Web page, however, these statements apply:

- You may specify **genProject**, which indicates a location for the associated Java-based objects that run in the Web application server
- If you do not specify **genProject**, all output is placed in the location identified in **genDirectory**

If you are generating from the EGL SDK, the following rules apply:

- You must specify **genDirectory**
- An error results if you specify **genProject**
- You cannot generate Java code for iSeries

#### Related concepts

"Generation from the EGL Software Development Kit (SDK)" on page 418

"Generation from the workbench batch interface" on page 417

"Generation in the workbench" on page 416

"Generation of Java code into a project" on page 409

## Related tasks

### Related reference

"Build descriptor options" on page 464

"buildPlan" on page 469

"genDirectory" on page 477

"prep" on page 485

"type in callLink element" on page 515

## genProperties

The build descriptor option **genProperties** specifies what kind of Java runtime properties to generate (if any) and, in some cases, whether to generate a linkage properties file. This build descriptor option is meaningful only when you are generating a Java program (which can use either kind of output) or a wrapper (which can use only the linkage properties file).

Valid values are as follows:

### NO (the default)

EGL does not generate runtime or linkage properties.

### PROGRAM

The effects are as follows:

- If you are generating a program to run outside of J2EE, EGL generates a properties file that is specific to the program being generated. The name of that file is as follows:  
*pgmAlias.properties*  
*pgmAlias*  
The name of the program at run time.
- The other effects occur whether you specify **PROGRAM** or **GLOBAL**:
  - If you are generating a program that runs in J2EE, EGL generates a J2EE environment file or into a deployment descriptor; for details, see *Understanding alternatives for setting deployment-descriptor values*.
  - If you are generating a Java wrapper or calling program, EGL may generate a linkage properties file; for details on the situation in which this file is generated, see *Linkage properties file (reference)*.

### GLOBAL

The effects are as follows:

- If you are generating a program to run outside of J2EE, EGL generates a properties file that is used throughout the run unit but is not named for the initial program in the run unit. The name of that properties file is **rununit.properties**.

This option is especially useful when the first program of a run unit does not access a file or database but calls programs that do.

When generating the caller, you can generate a properties file named for the program, and the content might include no database-related properties.

When you generate the called program, you can generate **rununit.properties**, and the content would be available for both programs.

- The other effects occur whether you specify **GLOBAL** or **PROGRAM**:
  - If you are generating a program that runs in J2EE, EGL generates a J2EE environment file or into a deployment descriptor; for details, see *Understanding alternatives for setting deployment-descriptor values*.

- If you are generating a Java wrapper or calling program, EGL may generate a linkage properties file; for details on the situation in which this file is generated, see *Linkage properties file (reference)*.

For further details, see *Java runtime properties* and *Linkage properties file*.

#### Related concepts

“J2EE environment file” on page 440  
 “Java runtime properties” on page 431  
 “Linkage options part” on page 399  
 “Linkage properties file” on page 447

#### Related tasks

“Setting deployment-descriptor values” on page 438

#### Related reference

“Build descriptor options” on page 464  
 “Java runtime properties (details)” on page 642

### genResourceBundle

When you are generating a VGWebTransaction program, a VGUI record, or a DataTable (type msgTable), the build descriptor option **genResourceBundle** specifies whether to generate a *Java resource bundle*, which is a Java object that contains strings to be presented at runtime. The content of a resource bundle generated for a VGUI record includes the Web-page title, labels, and help text; and the content of a resource bundle generated for a message table includes a set of messages.

Valid values of the option **genResourceBundle** are *yes* and *no*. Set the option to *yes* in any of these cases:

- You are generating a VGWebTransaction program and want a resource bundle for each VGUI record presented to the user
- You are generating only a VGUI record and want a resource bundle for it
- You are generating a DataTable that contains messages for presentation by a VGWebTransaction program

When you are generating a VGWebTransaction program and have specified a value for the program property **msgTablePrefix**, EGL produces a message resource bundle regardless of the value of **genResourceBundle**.

If you want to specify a Java locale for the generated resource bundle, set the build descriptor option **resourceBundleLocale**.

#### Related reference

“Build descriptor options” on page 464  
 “resourceBundleLocale” on page 486

### genVGUIRecords

When you are generating a VGWebTransaction program, the build descriptor option **genVGUIRecords** indicates whether you want to generate the VGUI records that are referenced in **converse** or **show** statements. Valid values are as follows:

#### yes (the default)

Generate the VGUI records.

- no** Do not generate the VGUI records. Use this setting only if the VGUI records referenced in the VGWebTransaction program were previously generated and have not changed since they were last generated.

For other details, see *VGUIRecord part*.

#### Related concepts

"Build descriptor part" on page 383

"VGUIRecord part" on page 167

#### Related reference

"Build descriptor options" on page 464

"converse" on page 672

"show" on page 751

"VGUIRecord part in EGL source format" on page 1089

### itemsNullable

The build descriptor option **itemsNullable** specifies the circumstance in which your code can set primitive fields to NULL.

Valid values are as follows:

#### NO (the default)

You cannot set primitive fields to NULL except in this case--

- The field is a primitive field in a non-fixed record or is a structure field in an SQL record; and
- The field-level property **isNullable** is set to yes.

This behavior is consistent with VisualAge Generator and with previous versions of EGL.

#### YES

You can set any of the following fields to NULL:

- A primitive variable, either standalone or in a non-fixed record
- A structure field in an SQL record

The behavior is consistent with the Informix product I4GL, but is not available for service parts or for interface parts of type basicInterface.

If a program invokes a function that is in an EGL library, both the program and library must be generated with the same setting for **itemsNullable**; otherwise, an error occurs when you compile the program.

The next table shows other effects of your decision.

*Table 9. Effect of itemsNullable*

Operation	ItemsNullable is set to NO	ItemsNullable is set to YES
Set a null field	Not possible for fields outside of an SQL record	You can set NULL by assigning an empty string ("") to a field

Table 9. Effect of **itemsNullable** (continued)

Operation	ItemsNullable is set to NO	ItemsNullable is set to YES
Test a null field	Not possible for fields outside of an SQL record	You can test NULL by testing the result of passing the string to the function StrLib.clip:  <pre>myString String = "";  // indicates that // the variable is NULL if (StrLib.clip(myString)     is NULL)     ; end</pre>
Assign a null field to another field	The value of the source is 0 or blank, and the assignment copies both a value and (if the target is nullable) the NULL state	If the target is nullable, the target is set to NULL. Otherwise, the target is set to 0 or blank
Concatenate a limited-length string with another string	The limited-length string is not padded	The limited-length string is padded with blanks to extend the string to the last position specified in the string declaration
Use a null field in a numeric expression	The field is treated as if it contained a 0	The expression evaluates to NULL
Use a null field in a text expression	The field is treated as if it contained a space	The field is treated as if it were an empty string
Use a null field in a logical expression	The expression is treated as if the value of the field were 0 or blank, with the next example evaluating to TRUE:  <pre>0 == null</pre>	The expression evaluates to TRUE only if null is compared with null, as is not the case in the next example, which evaluates to FALSE:  <pre>0 == null</pre>
SET™ field empty	Null state is not set	Null state is set
SET record empty	Null state is not set	Null state is set

### Related reference

“Build descriptor options” on page 464

## J2EE

The build descriptor option **J2EE** specifies whether a Java program is generated to run in a J2EE environment. Valid values are as follows:

### NO (the default)

Generates a program that will not run in a J2EE environment. The program connects to databases directly, and the environment is defined by a properties file.

### YES

Generates a program to run in a J2EE environment. The program connects to databases using a data source, and the environment is defined by a deployment descriptor.

When you generate a PageHandler, J2EE is always set to YES regardless of what is specified in this option. When you generate a Java program of type VGWebTransaction, J2EE is always set to NO regardless of what is specified in this option.

#### **Related concepts**

“EGL debugger” on page 369

#### **Related reference**

“Build descriptor options” on page 464

### **J2EELevel**

The build descriptor option **J2EELevel** specifies the J2EE level of the Web application server to which an EGL-generated Web service or service-binding library will be deployed.

Valid values are as follows:

- **1.4** (the default when the generated output will be deployed on WebSphere Application Server version 6.0)
- **1.3** (the default when the generated output will be deployed on WebSphere Application Server version 5.1)

This build descriptor option is used with option **serverType**.

#### **Related concepts**

“Build descriptor part” on page 383

“EGL services and Web services” on page 158

“Library part of type ServiceBindingLibrary” on page 172

#### **Related tasks**

“Generating in the workbench” on page 414

#### **Related reference**

“Build descriptor options” on page 464

“serverType” on page 487

### **linkage**

The build descriptor option **linkage** contains the name of the linkage options part that guides aspects of generation. This option is not required for generation, and no default value is available.

#### **Related concepts**

“Linkage options part” on page 399

#### **Related reference**

“Build descriptor options” on page 464

“callLink element” on page 499

### **msgTablePrefix**

When you generate a VGUI record bean alone, the build descriptor option **msgTablePrefix** specifies the message-table prefix that is stored in the VGUI record bean. When combined with an NLS code, that prefix determines the name of the DataTable that is the source of runtime messages. For details on the NLS code, see the topic for build descriptor option **targetNLS**.

When you generate a VGUI record bean along with a VGWebTransaction program, the value of the build descriptor option has no effect. Instead, the value of the program property **msgTablePrefix** determines what is stored in the VGUI record bean.

The value of **msgTablePrefix** has no effect on DataTable generation.

#### Related reference

"Build descriptor options" on page 464

"targetNLS" on page 495

### nextBuildDescriptor

The build descriptor option **nextBuildDescriptor** identifies the next build descriptor in chain, if any. For details, see *Build descriptor part*.

#### Related concepts

"Build descriptor part" on page 383

#### Related reference

"Build descriptor options" on page 464

### prep

The build descriptor option **prep** specifies whether EGL begins preparation when generation completes with a return code  $\leq 4$ . Valid values are YES and NO, and the default is YES.

Even if you set **prep** to NO, you can prepare code later. For details, see *Invoking a build plan after generation*.

Consider these cases:

- When you generate into a directory, EGL writes preparation messages to the directory specified in build descriptor option **genDirectory**, to the results file
- When you generate into a project (option **genProject**), the option **prep** has no effect, and preparation occurs in either of two situations:
  - Whenever you rebuild the project
  - Whenever you generate the source files; but only if you checked the workbench preference **Perform build automatically on resource modification**

If you wish to customize the generated build plan, do as follows:

- Set option **prep** to NO
- Set option **buildPlan** to YES (as is the default)
- Generate the output
- Customize the build plan
- Invoke the build plan, as described in *buildPlan*

#### Related concepts

"Results file" on page 414

#### Related tasks

"Invoking a build plan after generation" on page 419

#### Related reference

"Build descriptor options" on page 464

"buildPlan" on page 469



“Generated output (reference)” on page 626

“genDirectory” on page 477

“genProject” on page 478

## resourceAssociations

The build descriptor option **resourceAssociations** contains the name of a resource associations part, which relates record parts to files and queues on the target platforms. This option is not required for generation, and no default value is available.

### Related concepts

“Resource associations and file types” on page 393

### Related tasks

“Adding a resource associations part to an EGL build file” on page 397

### Related reference

“Build descriptor options” on page 464

“Association elements” on page 457

“Record and file type cross-reference” on page 860

## resourceBundleLocale

The build descriptor option **resourceBundleLocale** specifies a Java locale to be included in the name of a resource bundle that is generated for a VGUI record or for a message table, either of which is used in a VGWebTransaction program.

For other details, see *genResourceBundle*.

### Related reference

“Build descriptor options” on page 464

“genResourceBundle” on page 481

## secondaryTargetBuildDescriptor

The build descriptor option **secondaryTargetBuildDescriptor** specifies a build descriptor that guides the generation of code being deployed to a Web application server, while other code (which is targeted for other environments) is being generated in the same generation request.

Each of the following situations is of interest:

- You are generating a VGWebTransaction program with build descriptor option **genVGUIRecords** set to YES so that both the program (destined to run outside of a Web application server) and the VGUIRecords referenced by that program are generated
- You are selecting a project for generation when the EGL source folder contains the following parts, which may be in different packages:
  - PageHandlers to be deployed to a Web application server
  - Programs that the PageHandlers call from a platform that is outside of a Web application server

If you wish to use one build descriptor to control generation of two sets of parts, do this:

- Establish options for the parts that are not deployed to the Web application server
- Assign a value to the option **secondaryTargetBuildDescriptor** for the parts that are deployed to the Web application server.

In cases like those described earlier, **secondaryTargetBuildDescriptor** can reference a build descriptor to use when you generate the following output:

- VGUIRecords
- PageHandlers
- A message table that is produced during generation of a VGWebTransaction program
- A table that is used for validating VGUIRecord data, when the table is produced during generation of a VGWebTransaction program

You can set any options in the build descriptor named in **secondaryTargetBuildDescriptor**, but only the following have an effect:

- destDirectory
- destHost
- destPassword
- destPort
- destUserID
- genDirectory
- genProject
- genResourceBundle
- msgTablePrefix
- resourceBundleLocale
- system
- targetNLS
- tempDirectory

#### **Related concepts**

“Build descriptor part” on page 383

#### **Related tasks**

“Generating in the workbench” on page 414

#### **Related reference**

“Build descriptor options” on page 464

### **serverType**

The build descriptor option **serverType** identifies the type of Web Application Server in which your output will be deployed. The option is used to create appropriate output when you generate either an EGL-written Web service or a service binding library that binds to a Web service (which may or may not be written with EGL).

The appropriate setting depends on the Web application server on which you intend to deploy the generated output.

Valid values are as follows:

#### **WebSphereV6.0 (the default)**

The generated output will be deployed on WebSphere Application Server version 6.0.

### WebSphereV5.1

The generated output will be deployed on WebSphere Application Server version 5.1. (However, the minimum support required for Web Services is Websphere V5.1.1.)

This build descriptor option is used with option **J2EELevel**.

#### Related concepts

“Build descriptor part” on page 383

“EGL services and Web services” on page 158

“Library part of type ServiceBindingLibrary” on page 172

#### Related reference

“Build descriptor options” on page 464

“J2EELevel” on page 484

### serviceRuntime

The build descriptor option **serviceRuntime** specifies the runtime code that will be used to handle an EGL service, an EGL-generated Web service, or a service-binding library that references a Web service.

Valid values are as follows:

#### EGL (the default)

The deployed code is an EGL service, handled by EGL runtime.

#### WebSphere

The deployed code is handled by the WebSphere Web service runtime code, which is available when a Web service is deployed on WebSphere Application Server.

#### Related concepts

“Build descriptor part” on page 383

“EGL services and Web services” on page 158

“Library part of type ServiceBindingLibrary” on page 172

#### Related reference

“Build descriptor options” on page 464

### sessionBeanID

The build descriptor option **sessionBeanID** identifies the name of an existing session element in the J2EE deployment descriptor. The environment entries are placed into the session element when you act as follows:

- Generate a program for a Java platform (by setting **system** to AIX, WIN, or USS)
- Generate into an EJB project (by setting **genProject** to an EJB project)
- Request that environment properties be generated (by setting **genProperties** to GLOBAL or PROGRAM)

The option **sessionBeanID** is useful in the following case:

1. You generate a Java wrapper, along with an EJB session bean. In the EJB project deployment descriptor (file ejb-jar.xml), EGL creates a session element, without environment entries.

Both the EJB session bean and the session element are named as follows:

*ProgramnameEJBBean*

*Programname* is the name of the runtime program that receives data by way of the EJB session bean. The first letter in the name is uppercase, the other letters are lowercase.

In this example, the name of the program is ProgramA, and the name of the session element and the EJB session bean is ProgramaEJBBean.

2. After you generate the EJB session bean, you generate the Java program itself. Because the build descriptor option **genProperties** is set to YES, EGL generates J2EE environment entries into the deployment descriptor, into the session element established in step 1.
3. You generate ProgramB, which is a Java program that is used as a helper class for ProgramA. The values of **system** and **genProject** are the same as those used in step 2; also, you generate environment entries and set **sessionBeanID** to the name of the session element.

Your use of **sessionBeanID** causes EGL to place the environment entries for the second program into the session element that was created in step 2; specifically, into the session element ProgramaEJBBean.

In the portion of the deployment descriptor that follows, EGL created the environment entries **vgj.nls.code** and **vgj.nls.number.decimal** during step 2, when ProgramA was generated; but the entry **vgj.jdbc.default.database** is used only by ProgramB and was created during step 3:

```
<ejb-jar id="ejb-jar_ID">
  <display-name>EJBTest</display-name>
  <enterprise-beans>
    <session id="ProgramaEJBBean">
      <ejb-name>ProgramaEJBBean</ejb-name>
      <home>test.ProgramaEJBHome</home>
      <remote>test.ProgramaEJB</remote>
      <ejb-class>test.ProgramaEJBBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>vgj.nls.code</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>ENU</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>vgj.nls.number.decimal</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>.</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>vgj.jdbc.default.database</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>jdbc/Sample</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

A session element must be in the deployment descriptor before you can add environment entries. Because session elements are created during Java wrapper generation, it is recommended that you generate the Java wrapper before generating the related programs.

In the following cases, you generate a program into an EJB project, but the environment entries are placed into a J2EE environment file rather than into the deployment descriptor:

- **sessionBeanID** is set, but the session element that matches the value of **sessionBeanID** is not found in the deployment descriptor; or
- **sessionBeanID** is not set, and the session element that is named for the program is not found in the deployment descriptor. This case occurs when the program is generated before the wrapper.

For EJB projects, an environment entry name (like **vgj.nls.code**) can appear only once for each session element. If an environment entry already exists, EGL updates the entry type and value instead of creating a new entry.

EGL never deletes an environment entry from a deployment descriptor.

No default value is available for **sessionBeanID**.

#### Related reference

“Build descriptor options” on page 464

### sqlCommitControl

The build descriptor option **sqlCommitControl** allows generation of a Java runtime property that specifies whether a commit occurs after every change to the default database.

The property (`vgj.jdbc.default.database.autoCommit`) is generated only if the build descriptor option **genProperties** is also set to PROGRAM or GLOBAL. You can set the Java runtime property at deployment time regardless of your decision at generation time.

Valid values of **sqlCommitControl** are as follows:

#### NOAUTOCOMMIT

The commit is not automatic; the behavior is consistent with previous versions of EGL; and the Java runtime property is set to false, as is the default.

For details on the rules of commit and rollback in this case, see *Logical unit of work*.

#### AUTOCOMMIT

The commit is automatic; the behavior is consistent with previous versions of the Informix product I4GL; and the Java runtime property is set to true.

#### Related concepts

“Build descriptor options” on page 464

“Java runtime properties” on page 431

#### Related reference

“Build descriptor options” on page 464

“Default database” on page 298

“genProperties” on page 480

### sqlDB

The build descriptor option **sqlDB** specifies the default database used by a generated Java program that runs outside of J2EE. The value is a connection URL; for example, `jdbc:db2:MyDB`.

The option **sqlDB** is case-sensitive, has no default value, and is used only when you are generating a non-J2EE Java program. The option assigns a value to the Java runtime property **vgj.jdbc.default.database**, but only if option **genProperties** is set to GLOBAL or PROGRAM.

To specify the database used for validation , set **sqlValidationConnectionURL**.

#### Related concepts

"Java runtime properties" on page 431

"SQL support" on page 277

#### Related reference

"Build descriptor options" on page 464

"genProperties" on page 480

"Java runtime properties (details)" on page 642

"sqlPassword" on page 492

"sqlValidationConnectionURL" on page 493

"sqlJDBCDriverClass"

"validateSQLStatements" on page 497

### sqlID

The build descriptor option **sqlID** specifies a userid that is used to connect to a database during generation-time validation of SQL statements. You specify the database by setting **sqlValidationConnectionURL**.

When you generate a Java program, EGL also assigns the value of **sqlID** to the Java runtime property **vgj.jdbc.default.userid**. That property identifies the userid for connecting to the default database at run time, and you can specify the default database in **sqlDB**.

The option **sqlID** is case-sensitive and has no default value.

#### Related reference

"Build descriptor options" on page 464

"Java runtime properties (details)" on page 642

"sqlDB" on page 490

"sqlPassword" on page 492

"sqlValidationConnectionURL" on page 493

"sqlJDBCDriverClass"

"validateSQLStatements" on page 497

### sqlJDBCDriverClass

The build descriptor option **sqlJDBCDriverClass** specifies a driver class for connecting to the database that EGL uses to validate SQL statements at generation time. You specify the database by setting **sqlValidationConnectionURL**. Database access is through JDBC.

In the following cases EGL also assigns the value of **sqlJDBCDriverClass** to the Java runtime property **vgj.jdbc.drivers** in the program properties file:

- **genProperties** is set to GLOBAL or PROGRAM
- **J2EE** is set to NO

No default is available for the driver class, and the format varies by driver:

- For IBM DB2 APP DRIVER for Windows, the driver class is as follows--  
`COM.ibm.db2.jdbc.app.DB2Driver`

- For IBM DB2 NET DRIVER for Windows, the driver class is as follows--  
`COM.ibm.db2.jdbc.net.DB2Driver`
- For IBM DB2 UNIVERAL DRIVER for Windows, the driver class is as follows  
(with com in lower case)--  
`com.ibm.db2.jcc.DB2Driver`
- For the Oracle JDBC thin client-side driver, the driver class is as follows--  
`oracle.jdbc.driver.OracleDriver`
- For the IBM Informix JDBC driver, the driver class is as follows--  
`com.informix.jdbc.IfxDriver`

For other driver classes, refer to the documentation for the driver.

To specify more than one driver class, separate each class name from the next with a colon (:). You might do this if one Java program makes a local call to another but accesses a different database management system.

#### Related reference

“Build descriptor options” on page 464  
 “Informix and EGL” on page 299  
 “sqlDB” on page 490  
 “sqlID” on page 491  
 “sqlPassword”  
 “sqlValidationConnectionURL” on page 493  
 “validateSQLStatements” on page 497

### sqlJNDIName

The build descriptor option **sqlJNDIName** specifies the default database used by a generated Java program that runs in J2EE. The value is the name to which the default datasource is bound in the JNDI registry; for example, jdbc/MyDB.

The option **sqlJNDIName** is case-sensitive, has no default value, and is used only when you are generating a Java program for J2EE. The option assigns a value to the Java runtime property **vgj.jdbc.default.database**, but only if option **genProperties** is set to GLOBAL or PROGRAM.

To specify the database used for validation , set **sqlValidationConnectionURL**.

#### Related concepts

“Java runtime properties” on page 431  
 “SQL support” on page 277

#### Related reference

“Build descriptor options” on page 464  
 “genProperties” on page 480  
 “Java runtime properties (details)” on page 642  
 “sqlPassword”  
 “sqlValidationConnectionURL” on page 493  
 “sqlJDBCClass” on page 491  
 “validateSQLStatements” on page 497

### sqlPassword

The build descriptor option **sqlPassword** specifies a password that is used to connect to a database during generation-time validation of SQL statements. You specify the database by setting **sqlValidationConnectionURL**.

When you generate a Java program, EGL also assigns the value of **sqlPassword** to the Java runtime property **vgj.jdbc.default.password**. That property identifies the password for connecting to the default database at run time, and you can specify the default database in **sqlDB**.

The option **sqlPassword** is case-sensitive and has no default value.

#### Related concepts

"Java runtime properties" on page 431

#### Related reference

"Build descriptor options" on page 464

"Java runtime properties (details)" on page 642

"sqlDB" on page 490

"sqlID" on page 491

"sqlValidationConnectionURL"

"sqlJDBCDriverClass" on page 491

"validateSQLStatements" on page 497

### sqlValidationConnectionURL

The build descriptor option **sqlValidationConnectionURL** specifies a URL for connecting to the database that EGL uses to validate SQL statements at generation time. Database access is through JDBC.

No default is available for the URL, and the format varies by driver:

- For IBM DB2 APP DRIVER for Windows, the URL is as follows--

```
jdbc:db2:dbName
```

*dbName*

Database name

- For the Oracle JDBC thin client-side driver, the URL varies by database location. If the database is local to your machine, the URL is as follows--

```
jdbc:oracle:thin:dbName
```

If the database is on a remote server, the URL is as follows--

```
jdbc:oracle:thin:@host:port:dbName
```

*host*

Host name of the database server

*port*

Port number

*dbName*

Database name

- For the IBM Informix JDBC driver, the URL is as follows (with the lines combined into one)--

```
jdbc:informix-sqli://host:port  
/dbName:informixserver=servername;  
user=userName;password=password
```

*host*

Name of the machine on which the database server resides

*port*

Port number

*dbName*

Database name



*serverName*

Name of the database server

*userName*

Informix user ID

*passWord*

Password associated with the user ID

- For other drivers, refer to the documentation for the driver.

#### **Related reference**

"Build descriptor options" on page 464

"Informix and EGL" on page 299

"sqlDB" on page 490

"sqlID" on page 491

"sqlPassword" on page 492

"sqlJDBCClass" on page 491

"validateSQLStatements" on page 497

### **synchOnTrxTransfer**

The build descriptor option **synchOnTrxTransfer** specifies whether a commit point occurs when one the following kinds of programs runs a **transfer** statement of type *transfer to transaction*:

- A main program that participates in a batch or text application in a Java environment; or
- A main program of type z/OS batch or IMS BMP, as described in *Runtime configurations*.

Valid values are as follows:

#### **NO (the default)**

The **transfer** statement starts a program in the same run unit, but does not close or commit resources, which are available to the invoked program.

#### **YES**

The **transfer** statement commits recoverable resources, closes files, closes cursors, and starts a program in the same run unit.

#### **Related concepts**

"Build descriptor part" on page 383

"Runtime configurations" on page 11

"Run unit" on page 866

#### **Related reference**

"Build descriptor options" on page 464

"transfer" on page 752

### **system**

The build descriptor option **system** specifies the target platform for generation. This option is required; no default value is available. Valid values are as follows:

#### **AIX**

Indicates that generation produces a Java program that can run on AIX

#### **ISERIESJ**

Indicates that generation produces a Java program that can run on iSeries

## LINUX

Indicates that generation produces a Java program that can run on Linux (with an Intel processor)

## USS

Indicates that generation produces a Java program that can run on z/OS UNIX System Services

## WIN

Indicates that generation produces a Java program that can run on Windows 2000/NT/XP

### Related concepts

“Generated output” on page 625

“Linkage options part” on page 399

“Runtime configurations” on page 11

### Related reference

“Build descriptor options” on page 464

“callLink element” on page 499

“Generated output (reference)” on page 626

“Informix and EGL” on page 299

## targetNLS

The build descriptor option **targetNLS** specifies the national language code used to identify runtime messages. If the Java locale on the development machine is associated with one of the supported languages, the default value of **targetNLS** is the supported language. Otherwise, **targetNLS** has no default value.

The next table lists the supported languages. The code page for the language you specify must be loaded on your target platform.

Code	Languages
CHS	Simplified Chinese
CHT	Traditional Chinese
DES	Swiss German
DEU	German
ENP	Uppercase English (not supported on Windows 2000, Windows NT, and z/OS UNIX System Services)
ENU	US English
ESP	Spanish
FRA	French
ITA	Italian
JPN	Japanese
KOR	Korean
PTB	Brazilian Portuguese

For additional details on the message table used for programs, see the reference to the **messageTablePrefix** property in *Program part properties*. For additional details on the message table used for VGUI records in VGWebTransaction programs, see the topic for build descriptor option **msgTablePrefix**.

PageHandlers do not use a message table, but use a JavaServer Faces message resource. For details on that resource, see the description of the **msgResource** property in *PageHandler part in EGL source format*,

#### Related reference

“Build descriptor options” on page 464

“msgTablePrefix” on page 484

“PageHandler part in EGL source format” on page 785

“Program part properties” on page 856

### tempDirectory

When you generating a VGWebTransaction program or VGUIRecord into a Web project, the build descriptor option **tempDirectory** identifies the directory in which to store the JSP files. The name of each JSP file is as follows:

*recordAlias.jsp*

*recordAlias*

The name of the VGUI record or (if an alias is specified) the alias

The option **tempDirectory** is used only when a JSP file of the same name is in the WebContent\WEB-INF directory of the project to which output is generated.

If you do not specify a value for **tempDirectory** and if a JSP file with the name *recordAlias.jsp* is already in the directory WebContent\WEB-INF, the following statements apply:

- The new JSP file is stored with the following name, in the directory WebContent\WEB-INF:  
*newrecordAlias.jsp*
- Subsequent generations of the same UI record override the file *newrecordAlias.jsp*

A benefit of receiving the most recently generated JSP file is that you can copy and paste snippets of that file into the JSP file that you customized earlier. If you did not specify a value for **tempDirectory**, however, you should remove the file *newrecordAlias.jsp* from the Web project before deploying that project.

You use the build descriptor option **genProject** to direct output to a Web project. If you use the build descriptor option **genDirectory** instead, the effects are as follows:

- Output goes into a directory instead
- The option **tempDirectory** has no effect
- The prefix *new* is not in the name of the generated JSP file
- A generated JSP file overwrites any same-named file in the directory

#### Related concepts

#### Related reference

“Build descriptor options” on page 464

“genProject” on page 478

### userMessageFile

The build descriptor option **userMessageFile** specifies the generated value for Java runtime property **vgj.messages.file**, which specifies a properties file that includes messages you create or customize. The file is searched in these cases:

- When the EGL runtime responds to the invocation of function **SysLib.getMessage**, which returns a message that you created; for details, see *getMessage*

- When EGL runtime is handling a consoleUI application and attempts to present help or comment text from a file identified in the system variable **ConsoleLib.messageResource**, but that variable has no value.
- When EGL attempts to display a Java runtime message, as explained in *Message customization for EGL runtime messages*

The property is generated only if the value of build descriptor option **genProperties** is *yes* or *global*.

#### Related reference

“Build descriptor options” on page 464

“genProperties” on page 480

“getMessage()” on page 1032

“Message customization for EGL Java run time” on page 768

“messageResource” on page 910

## VAGCompatibility

The build descriptor option **VAGCompatibility** indicates whether the generation process allows use of special program syntax, as described in *Compatibility with VisualAge Generator*. Valid values are *no* and *yes*.

The setting of the EGL preference **VAGCompatibility** determines the default value of the build descriptor. If you are generating in the EGL SDK, no preferences are available, and the default value of **VAGCompatibility** is *no*.

Specify *yes* only if your program or PageHandler uses the special syntax.

#### Related concepts

“Build descriptor part” on page 383

“Compatibility with VisualAge Generator” on page 532

#### Related reference

“Build descriptor options” on page 464

## validateSQLStatements

The build descriptor option **validateSQLStatements** indicates whether SQL statements are validated against a database. Successful use of **validateSQLStatements** requires that you specify option **sqlValidationConnectionURL** and, in most cases, other options that begin with the letters **sql**, as listed later.

Valid values are YES and NO, and the default is NO. Validation of SQL statements increases the time required to generate your code.

When you request SQL validation, the database manager accessed from the generation platform prepares the SQL statements dynamically.

SQL statement validation has these restrictions:

- No validation is possible for SQL statements that use dynamic SQL and are based on SQL records
- The validation process may indicate errors that are found by the database manager in the generation environment but that will not be found by the database manager on the target platform

- Validation occurs only if your JDBC driver supports validation of SQL prepare statements and (in some cases) only if you have configured the driver to do such validation; for details, see the documentation for your JDBC driver

#### **Related reference**

"Build descriptor options" on page 464

"sqlID" on page 491

"sqlPassword" on page 492

"sqlValidationConnectionURL" on page 493

"sqlJDBCClass" on page 491

#### **webServiceEncodingStyle**

The build descriptor option **webServiceEncodingStyle** determines the format of the SOAP messages sent to and from an EGL-generated Web service. This option is used only for service generation.

For background information on Web Service Description Language (WSDL) style and encoding, see the following Web site:

<http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/>

Valid values are as follows:

##### **document/literal (the default)**

The transmitted data is handled in accordance with the document/literal wrapped style, as is usually appropriate.

##### **rpc/literal**

The transmitted data is handled in accordance with the rpc/literal style. Use this selection only if the client requires the rpc style and the literal encoding.

#### **Related concepts**

"EGL services and Web services" on page 158

#### **Related reference**

"Build descriptor options" on page 464

---

## **Build scripts**

### **Options required in EGL build scripts**

In EGL build scripts, certain preparation options are required if you are using DB2 UDB.

#### **Required options for DB2 precompiler**

The following options are required for DB2 usage and are included in the fdaptcl build script:

- HOST(COB2)
- APOSTSQL
- QUOTE

---

## callLink element

The callLink element of a linkage options part specifies the type of linkage used in a call. Each element includes these properties:

- pgmName
- type

The value of the **type** property determines what additional properties are available, as shown in the next sections:

- “If callLink type is localCall (the default)”
- “If callLink type is remoteCall”
- “If callLink type is ejbCall” on page 500

### If callLink type is localCall (the default)

Set property **type** to localCall when you are generating a Java program that calls a generated Java program that resides in the same thread. In this case, EGL middleware is not in use, and the following properties are meaningful for a callLink element in which **pgmName** identifies the called program--

- “alias in callLink element” on page 501
- “package in callLink element” on page 507
- “pgmName in callLink element” on page 509
- “type in callLink element” on page 515

You do not need to specify a callLink element for the call if the called program is in the same package as the caller and if either of these conditions is in effect:

- You do not specify an external name for the called program; or
- The external name for the called program is identical to the part name for that program.

The value of **type** cannot be localCall when you are generating a Java wrapper.

### If callLink type is remoteCall

Set property **type** to remoteCall when you are generating a Java program or wrapper, and the Java code calls a program that runs in a different thread. The call is not by way of a generated EJB session bean. In this case, EGL middleware is in use, and the following properties are meaningful for a callLink element in which **pgmName** identifies the called program--

- “alias in callLink element” on page 501
- “conversionTable in callLink element” on page 501
- “location in callLink element” on page 505
- “package in callLink element” on page 507 (used only if the generated code is calling a Java program that is stored in another package)
- “pgmName in callLink element” on page 509
- “remoteBind in callLink element” on page 510
- “remoteComType in callLink element” on page 511
- “remotePgmType in callLink element” on page 513
- “serverID in callLink element” on page 514
- “type in callLink element” on page 515

## If callLink type is ejbCall

Set property **type** to `ejbCall` when a `callLink` element is required to handle either of the following situations:

- You are generating a Java wrapper and intend to call the related, generated program by way of a generated EJB session bean
- You are generating a Java program and intend to call another generated program by way of a generated EJB session bean

In this case, EGL middleware is in use, and the following properties are meaningful for a `callLink` element in which **pgmName** identifies the called program:

- “alias in callLink element” on page 501
- “conversionTable in callLink element” on page 501
- “location in callLink element” on page 505
- “package in callLink element” on page 507 (used only if the generated Java code is calling a Java program that is stored in a package other than the package in which the EJB session bean resides)
- “parmForm in callLink element” on page 508 (used only if the generated Java code is calling a program that runs on CICS)
- “pgmName in callLink element” on page 509
- “providerURL in callLink element” on page 509
- “remoteBind in callLink element” on page 510
- “remoteComType in callLink element” on page 511
- “remotePgmType in callLink element” on page 513
- “serverID in callLink element” on page 514
- “type in callLink element” on page 515

### Related concepts

“Linkage options part” on page 399

“Runtime configurations” on page 11

### Related tasks

“Editing the callLink element of a linkage options part” on page 401

### Related reference

“alias in callLink element” on page 501

“conversionTable in callLink element” on page 501

“linkType in callLink element” on page 504

“location in callLink element” on page 505

“package in callLink element” on page 507

“pgmName in callLink element” on page 509

“providerURL in callLink element” on page 509

“remoteBind in callLink element” on page 510

“remoteComType in callLink element” on page 511

“remotePgmType in callLink element” on page 513

“serverID in callLink element” on page 514

“type in callLink element” on page 515

## alias in callLink element

The linkage options part, callLink element, property **alias** specifies the runtime name of the program identified in property **pgmName**. The property is meaningful only when **pgmName** refers to a program that is called by the program being generated.

The value of this property must match the alias (if any) you specified when declaring the program. If you did not specify an alias when declaring the program, either set the callLink element property **alias** to the name of the program part or do not set the property at all.

### Related concepts

"Linkage options part" on page 399

### Related tasks

"Editing the callLink element of a linkage options part" on page 401

### Related reference

"callLink element" on page 499

"pgmName in callLink element" on page 509

## conversionTable in callLink element

The linkage options part, callLink element, property **conversionTable** specifies the name of the conversion table that is used to convert data on a call. The property is meaningful only when **pgmName** identifies a program that is called by the generated program or wrapper.

The following details are in effect:

- When the call is to a non-Java program, a default conversion occurs in accordance with the character set (ASCII or EBCDIC) used on the calling platform. You must specify a value for **conversionTable** in the following case--
  - The caller is Java code and is on a machine that supports one character set (EBCDIC or ASCII); and
  - The called program is non-Java and is on a machine that supports the other character set.
- An attempt to specify a conversion table has no effect when EGL-generated Java code calls a Java program, except in the case of bidirectional text.
- The property **conversionTable** is available only if the value of property **type** is `ejbCall` or `remoteCall`.

Select one of the following values:

#### *conversion table name*

The caller uses the conversion table specified. For a list of tables, see *Data conversion*.

- \* Uses the default conversion table. The selected table is based either on the locale of the client machine or (if the client is running on a Web application server) on the locale of that server. If an unrecognized locale is found, English is assumed.

For a list of tables, see *Data conversion*.



**programControlled**

The caller uses the conversion table name that is in the system item `sysVar.callConversionTable` at run time. If `sysVar.callConversionTable` contains blanks, no conversion occurs.

**Related concepts**

"Linkage options part" on page 399

**Related tasks**

"Editing the callLink element of a linkage options part" on page 401

**Related reference**

"Bidirectional language text" on page 561

"callLink element" on page 499

"Data conversion" on page 558

"pgmName in callLink element" on page 509

"convert()" on page 1027

"targetNLS" on page 495

"type in callLink element" on page 515

## **ctgKeyStore in callLink element**

The linkage options part, callLink element, property **ctgKeyStore** is the name of the key store generated with the Java tool `keytool.exe` or with the CICS Transaction Gateway tool `IKEYMAN`. This property is required when the value of property **remoteComType** is set to `CICSSSL`.

**Related concepts**

"Linkage options part" on page 399

**Related reference**

"callLink element" on page 499

"ctgKeyStorePassword in callLink element"

"remoteComType in callLink element" on page 511

## **ctgKeyStorePassword in callLink element**

The linkage options part, callLink element, property **ctgKeyStorePassword** is the password used when generating the key store.

**Related concepts**

"Linkage options part" on page 399

**Related reference**

"callLink element" on page 499

"ctgKeyStore in callLink element"

"remoteComType in callLink element" on page 511

## **ctgLocation in callLink element**

The linkage options part, callLink element, property **ctgLocation** is the URL for accessing a CICS Transaction Gateway (CTG) server, as is used if the value of property **remoteComType** is `CICSECI` or `CICSSSL`. Specify the related port by setting the property **ctgPort**.

#### Related concepts

"Linkage options part" on page 399

#### Related reference

"callLink element" on page 499

"remoteComType in callLink element" on page 511

## ctgPort in callLink element

The linkage options part, callLink element, property **ctgPort** is the port for accessing a CICS Transaction Gateway (CTG) server, as is used if the value of property **remoteComType** is CICSECI or CICSSSL. Specify the related URL by setting the property **ctgLocation**.

If the case of CICSSSL, the value of **ctgPort** is the TCP/IP port on which a CTG JSSE listener is listening for requests; and if **ctgPort** is not specified, the CTG default port of 8050 is used.

#### Related concepts

"Linkage options part" on page 399

#### Related reference

"callLink element" on page 499

"ctgLocation in callLink element" on page 502

"remoteComType in callLink element" on page 511

## JavaWrapper in callLink element

The linkage options part, **callLink** element, property **javaWrapper** indicates whether to allow generation of Java wrapper classes that can invoke the program being generated.

Valid values are as follows:

#### No (the default)

Do not allow generation of Java wrapper classes.

#### Yes

Allow that generation to occur. The generation occurs only if the build descriptor option **enableJavaWrapperGen** is set to **yes** or **only**.

Your choice for **javaWrapper** property has an effect only when you are setting up a remote call, as occurs when the value of the **callLink** property **type** is **remoteCall**. In contrast, if you are setting up a call to the program by way of an EJB, the value of **javaWrapper** is always **yes**; and if you are setting up a local call, the value of **javaWrapper** is always **no**.

If you are generating in the workbench or from the workbench batch interface, the build descriptor option **genProject** identifies the project that receives the classes. If **genProject** is not specified (or if you are generating in the EGL SDK), the wrapper classes are placed in the directory specified by the build descriptor option **genDirectory**.

#### Related concepts

"Linkage options part" on page 399

#### Related reference

"callLink element" on page 499

"genDirectory" on page 477

"genProject" on page 478

## linkType in callLink element

The linkage options part, callLink element, property **linkType** specifies the type of linkage when the value of property **type** is localCall.

If you are generating a Java program, **linkType** is meaningful when property **pgmName** refers to a program that is called by the program being generated. If you are generating a Java wrapper, property type must be remoteCall or.ejbCall, and **linkType** is not available.

Select a value from this list:

#### DYNAMIC

Specifies that the call is to a Java program in the same thread. DYNAMIC is the default value .

#### STATIC

STATIC is equivalent to DYNAMIC unless you are using the Rational Application Developer for iSeries or the Rational Application Developer for z/OS..

#### Related concepts

"Linkage options part" on page 399

#### Related tasks

"Editing the callLink element of a linkage options part" on page 401

#### Related reference

"callLink element" on page 499

"pgmName in callLink element" on page 509

"type in callLink element" on page 515

## library in callLink element

The linkage options part, callLink element, property **library** specifies the DLL or library that contains the called program when the value of the **type** property is.ejbCall or remoteCall:

- If your EGL-generated Java program is calling a remote, non-EGL generated program on iSeries (for example, a C or C++ service program), the called program belongs to an iSeries library, and the **library** property refers to the name of the program that contains the entry point to be called. Set the other callLink properties as follows:
  - Set the **pgmName** property to the name of the entry point
  - Set the **remoteComType** property to direct or distinct
  - Set the **remotePgmType** property to externallyDefined
  - Set the **location** property to the name of the iSeries library
- Otherwise, if the calling program is an EGL-generated Java program not on iSeries, the **library** property refers to the name of a DLL that contains an entry point to be called locally as a native program. The entry point is identified by

the **pgmName** property; but you need to specify the **library** property only if the names of the entry point and DLL are different.

Do not include a file extension when you specify a value for the **library** property. For example, if the library name is *libxyz.so*, assign only *libxyz* to the property.

To call a native DLL, set the other callLink properties as follows:

- Set the **remoteComType** property to direct
- Set the **remotePgmType** property to externallyDefined
- Set the **type** property to remoteCall because EGL middleware is used even though the DLL is called on the machine where the Java program is running.

#### Related concepts

“Linkage options part” on page 399

#### Related reference

“callLink element” on page 499

## location in callLink element

The linkage options part, callLink element, property **location** specifies how the location of a called program is determined at run time. The property **location** is applicable in the following situation:

- The value of property **type** is *ejbCall* or *remoteCall*;
- The value of property **remoteComType** is *JAVA400*, *CICSECI*, *CICSSSL*, *CICSJ2C*, *IMS2J2C*, *IMSTCP*, or *TCPIP*; and
- One of these statements applies:
  - If you are generating a Java program, property **pgmName** refers to a program that is called by the program being generated
  - If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

Select a value from this list:

#### programControlled

Specifies that the location of the called program is obtained from the system function `sysVar.remoteSystemID` when the call occurs.

#### system name

Specifies the location where the called program resides.

If you are generating a Java program or wrapper, the meaning of this property depends on property **remoteComType**:

- If the value of **remoteComType** is *JAVA400*, **location** refers to the iSeries system identifier
- If the value of **remoteComType** is *CICSECI* or *CICSSSL*, **location** refers to the CICS system identifier
- If the value of **remoteComType** is *CICSJ2C*, **location** refers to the JNDI name of the ConnectionFactory object that you establish for the CICS transaction invoked by the call. You establish that ConnectionFactory object when setting up the J2EE server, as described in *Setting up the J2EE server for CICSJ2C calls*. By convention, the name of the ConnectionFactory object begins with *eis/*, as in the following example:

`eis/CICS1`

- If the value of **remoteComType** is IMSJ2C, **location** refers to the JNDI name of the ConnectionFactory object that you establish for the IMS transaction invoked by the call. You establish that ConnectionFactory object when setting up the J2EE server, as described in *Setting up the J2EE server for IMSJ2C calls*. By convention, the name of the ConnectionFactory object begins with eis/, as in the following example:

eis/IMS1

- If the value of **remoteComType** is IMSTCP, **location** has the following value:

*host:portNumber/dataStoreName*

*host*

TCP/IP host name or address that refers to the machine where the called program runs on IMS

*portNumber*

Number of the port used for TCP/IP connections by the target IMS Connect installation

*dataStoreName*

Target IMS datastore name, which must match the ID parameter of the Datastore element that is specified in the IMS Connect configuration member

- If the value of **remoteComType** is TCPIP, **location** refers to the TCP/IP host name or address and no default value exists
- If all the next conditions apply, **location** refers to the library of the called program--
  - The called program is an EGL-generated Java program that runs locally on iSeries
  - The value of **remoteComType** is DIRECT or DISTINCT
  - The value of **remotePgmType** is EXTERNALLYDEFINED

#### Related concepts

“Linkage options part” on page 399

#### Related tasks

“Editing the callLink element of a linkage options part” on page 401

“Setting up the J2EE server for CICSJ2C calls” on page 441

#### Related reference

“callLink element” on page 499

“pgmName in callLink element” on page 509

“remoteComType in callLink element” on page 511

“type in callLink element” on page 515

## luwControl in callLink element

The linkage options part, callLink element, property **luwControl** specifies whether the caller or called program controls the unit of work. This property is applicable only in the following situation:

- The value of property **type** is remoteCall; and
- You are generating a Java program or wrapper--
  - If you are generating a Java program, property **pgmName** refers to a CICS-based program that is called by the program being generated

- If you are generating a Java wrapper, **pgmName** refers to a CICS-based program that is called by way of the Java wrapper

Select one of the following values:

#### CLIENT

Specifies that the unit of work is under the caller's control. Updates by the called program are not committed or rolled back until the caller requests commit or rollback. If the called program issues a commit or rollback, a runtime error occurs.

CLIENT is the default value, unless a caller-controlled unit of work is not supported on the platform where the called program resides.

CLIENT is available if the caller is a Java wrapper or program that communicates with an iSeries-based COBOL program by way of the IBM Toolbox for Java. In this case, the value of **remoteComType** for the call is JAVA400.

#### SERVER

Specifies that a unit of work started by the called program is independent of any unit of work controlled by the calling program. In the called program, these rules apply:

- The first change to a recoverable resource begins a unit of work
- Use of the system functions sysLib.commit and sysLib.rollback are valid

On a call from EGL-generated Java code to a VisualAge Generator COBOL program, a commit (or rollback on abnormal termination) is issued automatically when the called program returns. That command affects only the changes that were made by the called program.

When the property **type** is ejbCall, the runtime behavior is as described for SERVER.

#### Related concepts

"Linkage options part" on page 399

"Logical unit of work" on page 395

#### Related tasks

"Editing the callLink element of a linkage options part" on page 401

#### Related reference

"callLink element" on page 499

"commit()" on page 1024

"rollback()" on page 1036

"pgmName in callLink element" on page 509

"type in callLink element" on page 515

## package in callLink element

The linkage options part, callLink element, property **package** identifies the Java package in which a called Java program resides. The property is useful whether property **type** is ejbcall, localCall, or remoteCall.

If you are generating a Java program, **package** is meaningful when property **pgmName** refers to a program that is called by the program being generated. If you are generating a Java wrapper, **package** is meaningful when property **pgmName** refers to the program that is called by way of the Java wrapper.

If the **package** property is not specified, the called program is assumed to be in the same package as the caller.

The package name that is used in generated Java programs is the package name of the EGL program, but in lower case; and when EGL generates output from the `callLink` element, the value of **package** is changed (if necessary) to lower case.

#### Related concepts

“Linkage options part” on page 399

#### Related tasks

“Editing the `callLink` element of a linkage options part” on page 401

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

#### Related reference

“`callLink` element” on page 499

“`pgmName` in `callLink` element” on page 509

“`type` in `callLink` element” on page 515

## parmForm in callLink element

The linkage options part, `callLink` element, property **parmForm** specifies the format of call parameters.

If you are generating a Java program, **parmForm** is applicable in this situation:

- Property **pgmName** refers to a CICS-based program that is called by the program being generated; and
- Property **type** is `ejbCall` or `remoteCall`; in either case, the valid **parmForm** values (as described later) are `COMMDATA` (the default) and `COMMPTR`.

If you are generating a Java wrapper, **parmForm** is applicable in this case:

- Property **pgmName** refers to a generated COBOL program that is called by way of the Java wrapper; and
- Property **type** is `ejbCall` or `remoteCall`; in either case, the valid **parmForm** values (as described later) are `COMMDATA` (the default) or `COMMPTR`.

Select a value from this list:

#### COMMDATA

Specifies that the caller places business data (rather than pointers to data) in the `COMMAREA`.

Each argument value is moved to the buffer adjoining the previous value without regard for boundary alignment.

`COMMDATA` is the default value if the property **type** is `ejbCall` or `remoteCall`.

#### COMMPTR

Specifies that the caller acts as follows:

- Places a series of 4-byte pointers in the `COMMAREA`, one pointer per argument passed
- Sets the high-order bit of the last pointer to 1

`COMMPTR` is the default value if the value of property **type** is `localCall`.

#### Related concepts

“Linkage options part” on page 399

### Related tasks

"Editing the callLink element of a linkage options part" on page 401

### Related reference

"callLink element" on page 499

"linkType in callLink element" on page 504

"parmForm in callLink element" on page 508

"pgmName in callLink element"

"type in callLink element" on page 515

## pgmName in callLink element

The linkage options part, callLink element, property **pgmName** specifies the name of the program part to which the callLink element refers.

You can use an asterisk (\*) as a global substitution character in the program name; however, that character is valid only as the last character. For details, see *Linkage options part*.

### Related concepts

"Linkage options part" on page 399

### Related tasks

"Editing the callLink element of a linkage options part" on page 401

### Related reference

"callLink element" on page 499

## providerURL in callLink element

The linkage options part, callLink element, property **providerURL** specifies the host name and port number of the name server used by an EGL-generated Java program or wrapper to locate an EJB session bean that in turn calls an EGL-generated Java program. The property must have the following format:

`iiop://hostName:portNumber`

*hostName*

The IP address or host name of the machine on which the name server runs

*portNumber*

The port number on which the name server listens

The property **providerURL** is applicable only in the following situation:

- The value of property **type** is `ejbCall`; and
- Property **pgmName** refers to the program being called from the Java program or wrapper being generated.

Enclose the URL in double quote marks to avoid a problem either with periods or with the colon that precedes the port number.

A default is used if you do not specify a value for **providerURL**. The default directs an EJB client to look for the name server that is on the local host and that listens on port 900. The default is equivalent to the following URL:

`"iiop://"`

The following **providerURL** value directs an EJB client to look for a remote name server that is called *bankserver.mybank.com* and that listens on port 9019:



"iiop://bankserver.mybank.com:9019"

The following property value directs an EJB client to look for a remote name server that is called *bankserver.mybank.com* and that listens on port 900:

"iiop://bankserver.mybank.com"

#### Related concepts

"Linkage options part" on page 399

#### Related tasks

"Editing the callLink element of a linkage options part" on page 401

"Setting up the J2EE runtime environment for EGL-generated code" on page 437

#### Related reference

"callLink element" on page 499

"pgmName in callLink element" on page 509

"type in callLink element" on page 515

## refreshScreen in callLink element

The linkage options part, callLink element, property **refreshScreen** indicates whether an automatic screen refresh is to occur when the called program returns control. Valid values are *yes* (the default) and *no*.

Set **refreshScreen** to *no* if the caller is in a run unit that presents text forms to a screen and either of these situations applies:

- The called program does not present a text form; or
- The caller writes a full-screen text form after the call.

The property **refreshScreen** applies only in these cases:

- The callLink **type** property is *localCall*; or
- The callLink **type** property is *remoteCall* when the *remoteComType* property is *direct* or *distinct*.

The property is ignored if you include the **noRefresh** indicator on the call statement.

#### Related reference

"call" on page 665

## remoteBind in callLink element

The linkage options part, callLink element, property **remoteBind** specifies whether linkage options are determined at generation time or at run time. This property is applicable only in the following situation:

- The value of property **type** is *ejbCall* or *remoteCall*; and
- You are generating a Java program or wrapper. The property **pgmName** may refer to a program that is called by the program being generated, in which case the entry refers to the call from program to program. Alternatively, the property may refer to the program being generated, in which case the entry refers to the call from wrapper to program.

Select one of these values:

## GENERATION

The linkage options specified at generation time are necessarily in use at run time. **GENERATION** is the default value.

## RUNTIME

The linkage options specified at generation time can be revised at deployment time. In this case, you must include a linkage properties file in the runtime environment.

EGL generates a linkage properties file in the following situation:

- You are generating a Java program or wrapper;
- You set the property **remoteBind** to **RUNTIME**; and
- You generate with the build descriptor option **genProperties** set to **GLOBAL** or **PROGRAM**.

### Related concepts

"Linkage options part" on page 399

"Linkage properties file" on page 447

### Related tasks

"Deploying a linkage properties file" on page 447

"Editing the callLink element of a linkage options part" on page 401

### Related reference

"callLink element" on page 499

"genProperties" on page 480

"Linkage properties file (details)" on page 764

"pgmName in callLink element" on page 509

"type in callLink element" on page 515

## remoteComType in callLink element

The linkage options part, callLink element, property **remoteComType** specifies the communication protocol used in the following case:

- The value of property **type** is **ejbCall** or **remoteCall**; and
- You are generating a Java program or wrapper--
  - If you are generating a Java program, property **pgmName** refers to a program that is called by the program being generated
  - If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

Select one of the following values.

### DEBUG

Causes the called program to run in the EGL debugger, even when the calling program is running in a Java runtime or Java debug environment. You might use this setting in the following cases:

- You are running a Java program that uses an EGL Java wrapper to call a program written with EGL; or
- You are running an EGL-generated calling program that calls a program written with EGL.

The preceding situations can occur outside the WebSphere Test Environment, but can also occur within that environment, as when a JSP invokes a program written with EGL. In any case, the effect is to invoke the EGL source, not an EGL-generated program.

If you are using the WebSphere Test Environment, the caller and called programs must both be running there; the call cannot be from a remote machine.

When you use DEBUG, you set the following properties in the same **callLink** element--

- **library**, which names the project that contains the called program
- **package**, which identifies the package that contains the called program; but you do not need to set this property if the caller and called programs are in the same package

If the caller is not running in the EGL debugger and is not running in the WebSphere Test Environment, you must set these properties of the **callLink** element:

- **serverid**, which should specify the listener's port number if it's not 8346; and
- **location**, which must contain the hostname of the machine where the Eclipse workbench is running.

### DIRECT

Specifies that the calling program or wrapper uses a direct local call, which means that the calling and called code run in the same thread. No TCP/IP listener is involved, and the value of property **location** is ignored. DIRECT is the default.

A calling Java program does not use the EGL middleware, but a calling wrapper uses that middleware to handle data conversion between EGL and Java primitive types.

If the EGL-generated Java code is calling a non-EGL-generated dynamic link library (DLL) or a C or C++ program, it is recommended that you use the **remoteComType** value DISTINCT.

### DISTINCT

Specifies that a new run unit is started when calling a program locally. The call is still considered to be remote because EGL middleware is involved.

You can use this value for an EGL-generated Java program that calls a dynamic link library (DLL) or a C or C++ program.

### CICSECI

Specifies use of the CICS Transaction Gateway (CTG) ECI interface, as is needed when you are debugging or running non-J2EE code that accesses CICS.

CTG Java classes are used to implement this protocol. To specify the URL and port for a CTG server, assign values to the **callLink** element, properties **ctgLocation** and **ctgPort**. To identify the CICS region where the called program resides, specify the **location** property.

### CICSJ2C

Specifies use of a J2C connector for the CICS Transaction Gateway.

### CICSSSL

Specifies use of the Secure Socket Layer (SSL) features of CICS Transaction Gateway (CTG). The JSSE implementation of SSL is supported.

CTG Java classes are used to implement this protocol. To specify additional information for a CTG server, assign values to the following **callLink** element properties:

- **ctgKeyStore**

- `ctgKeyStorePassword`
- `ctgLocation`
- `ctgPort`, which in this case is the TCP/IP port on which a CTG JSSE listener is listening for requests. If `ctgPort` is not specified, the CTG default port of 8050 is used.

To identify the CICS region where the called program resides, specify the `location` property.

### IMSJ2C

In a J2EE environment, specifies use of an IMS J2C connector.

### IMSTCP

Specifies use of the TCP/IP connector from IMS Connect. Select this option if you are running non-J2EE code that accesses IMS.

**Note:** IMS Connect must be installed on the host machine where the IMS program resides. On the client machine, the following set of IMS Connector for Java jar files must be in the classpath when the calling program is started:

- `ccf2.jar`
- `connector.jar`
- `imsico.jar`

### JAVA400

Specifies use of the IBM Toolbox for Java to communicate between a Java wrapper or program and a COBOL program that was generated (by EGL or VisualAge Generator) for iSeries.

### TCPIP

Specifies that the EGL middleware uses TCP/IP.

### Related concepts

“Linkage options part” on page 399

### Related tasks

“Editing the `callLink` element of a linkage options part” on page 401

“Setting up the J2EE server for CICSJ2C calls” on page 441

“Setting up the TCP/IP listener for a called appl in a J2EE appl client module” on page 442

“Setting up the TCP/IP listener for a called non-J2EE application” on page 436

### Related reference

“`callLink` element” on page 499

“`ctgKeyStore` in `callLink` element” on page 502

“`ctgKeyStorePassword` in `callLink` element” on page 502

“`ctgLocation` in `callLink` element” on page 502

“`ctgPort` in `callLink` element” on page 503

## remotePgmType in callLink element

The linkage options part, `callLink` element, property **`remotePgmType`** specifies the kind of program being called. The property is applicable in the following situation:

- The value of property **`type`** is `ejbCall` or `remoteCall`; and
- One of these statements applies:
  - If you are generating a program (rather than a wrapper), property **`pgmName`** refers to a program that is called by the program being generated.

The called program is one of the following kinds:

- An EGL-generated Java program
- A non-EGL-generated dynamic link library (DLL) or C or C++ program
- A program that runs on CICS and has CICS commands
- A program that runs on IMS
- If you are generating a Java wrapper, **pgmName** refers to the program that is called by way of the Java wrapper.

#### EGL

The called program is a COBOL or Java program that was generated by EGL or by VisualAge Generator; in this case, the caller is a Java program or Java wrapper. This value is the default.

#### EXTERNALLYDEFINED

The called program was generated neither by EGL nor by VisualAge Generator. This option is available only in the following cases:

- The caller is a Java program, and the called program runs on CICS and includes CICS commands.
- Alternatively, the caller is a Java program, and the called program is either an EGL-generated Java program or a non-EGL-generated DLL or a C or C++ program.

If the caller is invoking a DLL, set the library property to the name of the shared library if it isn't the same as the name of the function being called within the DLL

#### Related concepts

"Linkage options part" on page 399

"Runtime configurations" on page 11

#### Related tasks

"Editing the callLink element of a linkage options part" on page 401

#### Related reference

"callLink element" on page 499

"library in callLink element" on page 504

"pgmName in callLink element" on page 509

"type in callLink element" on page 515

## serverID in callLink element

The linkage options part, callLink element, property **serverID** specifies one of the following values:

- The TCP/IP port number of a called program's listener; but only if the TCP/IP protocol is in use. In this case, no default exists.
- The ID of a CICS transaction being invoked, but only when access to CICS is by the ECI interface or Secure Socket Layer features of the CICS Transaction Gateway. In this case, the default is the CICS server system mirror transaction.
- The ID of an IMS transaction being invoked, but only when the value of property **remoteComType** is IMSTCP.

The property is used only in the following situation:

- The value of property **type** is **ejbCall** or **remoteCall**;
- The value of **remoteComType** is **TCPIP**, **CICSECI**, **CICSSSL**, or **IMSTCP**; and
- You are generating a Java program or wrapper--

- If you are generating a Java program, property **pgmName** refers to a program that is called by the program being generated
- If you are generating a Java wrapper, **pgmName** refers to a program that is called by way of the Java wrapper

#### Related concepts

“Linkage options part” on page 399

#### Related tasks

“Editing the callLink element of a linkage options part” on page 401

“Setting up the TCP/IP listener for a called appl in a J2EE appl client module” on page 442

“Setting up the TCP/IP listener for a called non-J2EE application” on page 436

#### Related reference

“callLink element” on page 499

“pgmName in callLink element” on page 509

“remoteComType in callLink element” on page 511

“type in callLink element”

## type in callLink element

The linkage options part, callLink element, property **type** specifies the kind of call. Select one of the following values:

### ejbCall

Indicates that the generated Java program or wrapper will implement the program call by using an EJB session bean and that the EJB session bean will access program identified in the property **pgmName**. The value **ejbCall** is applicable in either of two cases:

- You are generating a Java wrapper and the wrapper calls that program by way of an EJB session bean. In this case, the property **pgmName** refers to the program called from the wrapper, and your use of **ejbCall** causes generation of the EJB session bean.
- You are generating a Java program that calls a generated program by way of an EJB session bean. In this case, the property **pgmName** refers to the called program, and an EJB session bean is not generated.

In either case, if you are using an EJB session bean, you must generate a Java wrapper, if only to generate the EJB session bean.

The generated session bean must be deployed on an enterprise Java server, and one of the following statements must be true:

- The name server used to locate the EJB session bean resides on the same machine as the code calling that session bean; or
- The property **providerURL** identifies where the name server resides.

If you wish to use an EJB session bean, you must generate the calling program or wrapper with a linkage options part in which the value of property **type** for the called program is **ejbCall**. You cannot make the decision to use a session bean at deployment time. If you set the property **remoteBind** to **RUNTIME**, however, you can decide at deployment time *how* the EJB session bean accesses the generated program, although making this decision at generation time is more efficient.

### localCall

Specifies that the call does *not* use EGL middleware. The called program in this case is in the same process as the caller.

**localCall** is the default value

#### **remoteCall**

Specifies that the call uses EGL middleware, which adds 12 bytes to the end of the data passed. Those bytes allow the caller to receive a return value from the called program.

If the caller is Java code, communication is handled by the protocol specified in property **remoteComType**; the protocol choice indicates whether the called program is in the same or a different thread.

If a fixed record with a variable length is passed on a call, these statements apply:

- Space is reserved for the maximum length specified for the record
- If the value of callLink property **type** is remoteCall or ejbCall, the variable-length item (if any) must be inside the record

#### **Related concepts**

"Linkage options part" on page 399

#### **Related tasks**

"Editing the callLink element of a linkage options part" on page 401

#### **Related reference**

"callLink element" on page 499

"linkType in callLink element" on page 504

"location in callLink element" on page 505

"parmForm in callLink element" on page 508

"pgmName in callLink element" on page 509

"providerURL in callLink element" on page 509

"remoteComType in callLink element" on page 511

---

## **C functions with EGL**

EGL programs can invoke C functions.

#### **To invoke a C function from EGL:**

After you have identified the C functions to use in your EGL program, you must:

1. Download the EGL stack library and application object file from the IBM website to your computer.
2. Compile all C code into one shared library and link it with the appropriate platform-specific stack library.
3. Create a function table.
4. Compile the function table and the appropriate platform-specific application object file into a shared library, and link this shared library with the shared library created in Step 2 and the stack library.

#### **1. Download the EGL stack library and application object file**

To download the EGL stack library and application object file:

1. Locate the EGL Support website.
  - The URL for Rational Application Developer is:  
<http://www3.software.ibm.com/ibmdl/pub/software/rational/sdp/rad/60/redist>
  - The URL for Rational Web Developer is:

<http://www3.software.ibm.com/ibmdl/pub/software/rational/sdp/rwd/60/redist>

2. Download the **EGLRuntimesV60IFix001.zip** file to your preferred directory.
3. Unzip **EGLRuntimesV60IFix001.zip** to identify the following files:

For the platform-specific stack libraries:

- AIX: EGLRuntimes/Aix/bin/libstack.so
- Linux: EGLRuntimes/Linux/bin/libstack.so
- Win32:
  - EGLRuntimes/Win32/bin/stack.dll
  - EGLRuntimes/Win32/bin/stack.lib

For the platform-specific application object files:

- AIX: EGLRuntimes/Aix/bin/application.o
- Linux: EGLRuntimes/Linux/bin/application.o
- Win32: EGLRuntimes/Win32/bin/application.obj

## 2. Compile all C code into a shared library

Your C code receives values from EGL using pop external functions and returns values to EGL using return external functions. The pop external functions are described in *Receiving values from EGL*; the return external functions are described in *Returning values to EGL*.

To compile all C code into a shared library:

1. Using standard methods, compile all of your C code into one shared library and link it with the appropriate platform-specific EGL stack library.
2. In the following platform-specific examples, **file1.c** and **file2.c** are C files containing functions invoked from EGL.

On AIX (the `ld` command must be on a single line):

```
cc -c -Iincl_dir file1.c file2.c
ld -G -b32 -bexpall -bnoentry
    -brtl file1.o file2.o -Lstack_lib_dir
    -lstack -o lib1_name -lc
```

On Linux (the `gcc` command must be on a single line):

```
cc -c -Iincl_dir file1.c file2.c
gcc -shared file1.o file2.o -Lstack_lib_dir
    -lstack -o lib1_name
```

On Windows (the `link` command must be on a single line):

```
cl /c -Iincl_dir file1.c file2.c
link /DLL file1.obj file2.obj
    /LIBPATH:stack_lib_dir
    /DEFAULTLIB:stack.lib /OUT:lib1_name
```

*incl\_dir*

the directory location for the header files.

*stack\_lib\_dir*

the directory location for the stack library.



*lib1\_name*  
the name of the output library.

**Note:** If your C code is using any of the IBM Informix ESQL/C library functions (BIGINT, DECIMAL, DATE, INTERVAL, DATETIME), then the ESQL/C library must also be linked.

### 3. Create a function table

The function table is a C source file which includes the names of all C functions to be invoked from the EGL program. In the following function table example, **c\_fun1** and **c\_fun2** are names of the C functions. All of the functions identified in the code must have been exported from the C shared library created in Step 2 above.

```
#include <stdio.h>
struct func_table {

    char *fun_name;
    int (*fptr)(int);
};

extern int c_fun1(int);
extern int c_fun2(int);
/* Similar prototypes for other functions */

struct func_table ftab[] =
{
    "c_fun1", c_fun1,
    "c_fun2", c_fun2,
    /* Similarly for other functions */
    "", NULL
};
```

Create a function table based on the example above, and populate the function table with the appropriate C functions. Indicate the end of the function table with **""**, **NULL**.

### 4. Compile the function table and the platform-specific application object file into a shared library

The application object file is the interface between the EGL code and the C code.

The following two artifacts must be compiled into one shared library and linked with the stack library and the library created in Step 2 above:

- function table
- application object file

Compile the new shared library using the following example, where **ftable.c** is the name of the function table and **mylib** is the name of the C shared library created in Step 2 and **lib\_dir** is the directory location for **mylib**. Specify **lib2\_name** by using the *dllName* property or the *vgj.defaultI4GLNativeLibrary* Java runtime property.

On AIX (the **ld** command must be on a single line):

```
cc -c ftable.c
ld -G -b32 -bexpall -bnoentry
    -brtl ftable.o application.o
    -lstack -lib_dir -lstack -llib_dir
    -lmylib -o lib2_name -lc
```

On Linux (the gcc command must be on a single line):

```
cc -c ftable.c
gcc -shared ftable.o application.o
    -lstack_lib_dir -lstack -llib_dir
    -lmylib -o lib2_name
```

On Windows (the link command must be on a single line):

```
cl /c ftable.c
link /DLL ftable.obj application.obj
    /LIBPATH:stack_lib_dir
    /DEFAULTLIB:stack.lib
    /LIBPATH:lib_dir
    /DEFAULTLIB:mylib.lib /OUT:lib2_name
```

Link the three libraries together.

With your C shared library, function table, and stack library linked, you are now ready to invoke the C functions from your EGL code. For information on how to invoke a C function in EGL, see *Invoking a C function from an EGL program*.

#### Related concept

“Linkage options part” on page 399

#### Related reference

“BIGINT functions for C”

“C data types and EGL primitive types” on page 520

“DATE functions for C” on page 521

“DATETIME and INTERVAL functions for C” on page 522

“DECIMAL functions for C” on page 523

“Invoking a C Function from an EGL Program” on page 524

“Return functions for C” on page 528

“Stack functions for C” on page 526

## BIGINT functions for C

**Note:** The following BIGINT functionality is available only to users of IBM Informix ESQL/C. To use these functions, ESQL/C users will need to manually link their C code to the ESQL/C libraries.

The BIGINT data type is a machine-independent method for representing numbers in the range of  $-2^{63}-1$  to  $2^{63}-1$ . ESQL/C provides routines that facilitate the conversion from the BIGINT data type to other data types in the C language.

The BIGINT data type is internally represented with the **ifx\_int8\_t** structure. Information about the structure can be found in the header file **int8.h**, which is included in the ESQL/C product. Include this file in all C source files that use any of the BIGINT functions.

All operations on **int8** type numbers must be performed using the following ESQL/C library functions for the **int8** data type. Any other operations, modifications, or analyses can produce unpredictable results. The ESQL/C library provides the following functions that allow you to manipulate **int8** numbers and convert **int8** type numbers to and from other data types.

Function Name	Description
ifx_int8add( )	Adds two BIGINT type values

Function Name	Description
ifx_int8cmp( )	Compares two BIGINT type numbers
ifx_int8copy( )	Copies an <b>ifx_int8_t</b> structure
ifx_int8cvasc( )	Converts a C <b>char</b> type value to a BIGINT type number
ifx_int8cvdbl( )	Converts a C <b>double</b> type number to a BIGINT type number
ifx_int8cvdec( )	Converts a <b>decimal</b> type value into a BIGINT type value
ifx_int8cvflt( )	Converts a C <b>float</b> type value into a BIGINT type value
ifx_int8cvint( )	Converts a C <b>int</b> type number into a BIGINT type number
ifx_int8cvlong( )	Converts a C <b>long</b> ( <b>int</b> on 64 bit machine) type value to a BIGINT type value
ifx_int8cvlong_long( )	Converts a C <b>long long</b> type (8-byte value, <b>long long</b> in 32 bit and <b>long</b> in 64 bit) value into a BIGINT type value
ifx_int8div( )	Divides two BIGINT numbers
ifx_int8mul( )	Multiplies two BIGINT numbers
ifx_int8sub( )	Subtracts two BIGINT numbers
ifx_int8toasc( )	Converts a BIGINT type value to a C <b>char</b> type value
ifx_int8todbl( )	Converts a BIGINT type value to a C <b>double</b> type value
ifx_int8todec( )	Converts a BIGINT type number into a <b>decimal</b> type number
ifx_int8toflt( )	Converts a BIGINT type number into a C <b>float</b> type number
ifx_int8toint( )	Converts a BIGINT type value to a C <b>int</b> type value
ifx_int8tolong( )	Converts a BIGINT type value to a C <b>long</b> ( <b>int</b> on 64 bit machine) type value
ifx_int8tolong_long( )	Converts a C <b>long long</b> ( <b>long</b> on 64 bit machine) type to a BIGINT type value

#### Related reference

For more information about the individual functions, see the following:  
IBM Informix ESQL/C Programmer's Manual.

"DATE functions for C" on page 521

"DATETIME and INTERVAL functions for C" on page 522

"DECIMAL functions for C" on page 523

"Invoking a C Function from an EGL Program" on page 524

## C data types and EGL primitive types

The following table shows the mapping between C data types, I4GL data types, and EGL primitive types.

C data types	Equivalent I4GL data type	Equivalent EGL primitive type
char	CHAR or CHARACTER	UNICODE(1)
char	NCHAR	UNICODE(size)
char	NVARCHAR	STRING
char	VARCHAR	STRING
int	INT or INTEGER	INT

C data types	Equivalent I4GL data type	Equivalent EGL primitive type
short	SMALLINT	SMALLINT
ifx_int8_t	BIGINT	BIGINT
dec_t	DEC or DECIMAL(p,s,) or NUMERIC(p)	DECIMAL(p)
dec_t	MONEY	MONEY
double	FLOAT	FLOAT
float	SMALLFLOAT	SMALLFLOAT
loc_t	TEXT	CLOB
loc_t	BYTE	BLOB
int	DATE	DATE
dtime_t	DATETIME	TIMESTAMP
intvl_t	INTERVAL	INTERVAL

### Related reference

“BIN and the integer types” on page 50

“BLOB” on page 49

“CLOB” on page 48

“DATE” on page 41

“DECIMAL” on page 50

“FLOAT” on page 51

“INTERVAL” on page 42

“Invoking a C Function from an EGL Program” on page 524

“MBCHAR” on page 39

“MONEY” on page 51

“NUM” on page 51

“Primitive types” on page 34

“SMALLFLOAT” on page 53

“TIME” on page 44

“TIMESTAMP” on page 44

## DATE functions for C

**Note:** The following DATE functionality is available only to users of IBM Informix ESQL/C. To use these functions, ESQL/C users will need to manually link their C code to the ESQL/C libraries.

The following date-manipulation functions are in the ESQL/C library. They convert dates between a string format and the internal DATE format.

Function Name	Description
rdatestr( )	Converts an internal DATE to a character string format
rdayofweek( )	Returns the day of the week of a date in internal format
rdefmtdate( )	Converts a specified string format to an internal DATE
rfmtdate( )	Converts an internal DATE to a specified string format
rjulmdy( )	Returns month, day, and year from a specified DATE
rleapyear( )	Determines whether the specified year is a leap year

Function Name	Description
rmdayjul( )	Returns an internal DATE from month, day, and year
rstrdate( )	Converts a character string format to an internal DATE
rtoday( )	Returns a system date as an internal DATE

### Related reference

For more information about the individual functions, see the following:  
IBM Informix ESQL/C Programmer's Manual.

"BIGINT functions for C" on page 519

"DATETIME and INTERVAL functions for C"

"DECIMAL functions for C" on page 523

"Invoking a C Function from an EGL Program" on page 524

## DATETIME and INTERVAL functions for C

**Note:** The following DATETIME and INTERVAL functionality is available only to users of IBM Informix ESQL/C. To use these functions, ESQL/C users will need to manually link their C code to the ESQL/C libraries.

The DATETIME and INTERVAL data types are internally represented with the **dtime\_t** and **intrvl\_t** structures, respectively. Information about these structures can be found in the header file **datetime.h**, which is included in the ESQL/C product. Include this file in all C source files that use any of the DATETIME and INTERVAL functions.

You must use the following ESQL/C library functions for the **datetime** and **interval** data types to perform all operations on those types of values.

Function Name	Description
dtaddinv( )	Adds an interval value to a datetime value
dtcurrent( )	Gets the current date and time
dtcvasc( )	Converts an ANSI-compliant character string to a datetime value
dtcvfmtasc( )	Converts a character string with a specified format to a datetime value
dtextend( )	Changes the qualifier of a datetime value
dtsub( )	Subtracts one datetime value from another
dsubinv()	Subtracts an interval value from a datetime value
dttoasc( )	Converts a datetime value to an ANSI-compliant character string
dttofmtasc( )	Converts a datetime value to a character string with a specified format
incvasc( )	Converts an ANSI-compliant character string to an interval value
incvfmtasc( )	Converts a character string with a specified format to an interval value
intoasc( )	Converts an interval value to an ANSI-compliant character string

Function Name	Description
intofmtasc( )	Converts an interval value to a character string with a specified format
invdivdbl( )	Divides an interval value by a numeric value
invdivinv( )	Divides an interval value by another interval value
invextend( )	Extends an interval value to a different interval qualifier
invmuldbl( )	Multiplies an interval value by a numeric value

### Related reference

For more information about the individual functions, see the following:  
IBM Informix ESQL/C Programmer's Manual.

"BIGINT functions for C" on page 519

"DATE functions for C" on page 521

"DECIMAL functions for C"

"Invoking a C Function from an EGL Program" on page 524

## DECIMAL functions for C

**Note:** The following DECIMAL functionality is available only to users of IBM Informix ESQL/C. To use these functions, ESQL/C users will need to manually link their C code to the ESQL/C libraries.

The data type DECIMAL is a machine-independent method for representing numbers of up to 32 significant digits, with or without a decimal point, and with exponents in the range -128 to +126. ESQL/C provides routines that facilitate the conversion of DECIMAL-type numbers to and from every data type allowed in the C language. DECIMAL-type numbers consist of an exponent and a mantissa (or fractional part) in base 100. In normalized form, the first digit of the mantissa must be greater than zero.

The DECIMAL data type is internally represented with the **dec\_t** structure. The **decimal** structure and the type definition **dec\_t** can be found in the header file **decimal.h**, which is included in the ESQL/C product. Include this file in all C source files that use any of the decimal functions.

All operations on **decimal** type numbers must be performed using the following ESQL/C library functions for the **decimal** data type. Any other operations, modifications or analyses can produce unpredictable results.

Function Name	Description
deccvasc( )	Converts C <b>int1</b> type to DECIMAL type
dectoasc( )	Converts DECIMAL type to C <b>int1</b> type
deccvint( )	Converts C <b>int</b> type to DECIMAL type
dectoint( )	Converts DECIMAL type to C <b>int</b> type
deccvlong( )	Converts C <b>int4</b> type to DECIMAL type
dectolong( )	Converts DECIMAL type to C <b>int4</b> type
deccvflt( )	Converts C <b>float</b> type to DECIMAL type

Function Name	Description
dectoflt( )	Converts DECIMAL type to C <b>float</b> type
deccvdbl( )	Converts C <b>double</b> type to DECIMAL type
dectodbl( )	Converts DECIMAL type to C <b>double</b> type
decadd( )	Adds two DECIMAL numbers
decsub( )	Subtracts two DECIMAL numbers
decmul( )	Multiplies two DECIMAL numbers
decdiv( )	Divides two DECIMAL numbers
deccmp( )	Compares two DECIMAL numbers
deccopy( )	Copies a DECIMAL number
dececv( )	Converts DECIMAL value to ASCII string
decfcvt( )	Converts DECIMAL value to ASCII string

### Related reference

For more information about the individual functions, see the following:  
 IBM Informix ESQL/C Programmer's Manual.  
 "BIGINT functions for C" on page 519  
 "DATE functions for C" on page 521  
 "DATETIME and INTERVAL functions for C" on page 522  
 "Invoking a C Function from an EGL Program"

## Invoking a C Function from an EGL Program

You can invoke (or call) a C function from an EGL program. Prior to following the instructions below, you must compile and link your C code as identified in *C functions with EGL*.

To invoke a C function from an EGL program:

1. Using the *function invocation* statement, specify the following:
  - The name of the C function
  - Any arguments to pass to the C function
  - Any variables to return to the EGL program
2. Create an EGL native *library part* containing the function definition.
3. With the USE statement, specify the EGL native library part in the calling module.

For example, the following function invocation statement calls the C function **sendmsg( )**

```
sendmsg(chartype, 4, msg_status, return_code);
```

It passes two arguments (**chartype** and **4**, respectively) to the function and expects two arguments to be passed back (**msg\_status** and **return\_code**, respectively). This is made clear by defining the function in a native library as follows:

```
Library I4GLFunctions type nativeLibrary
{callingConvention = "I4GL", dllName = "mydll"}
  Function sendmsg(chartype char(10) in, i int in, msg_status int out, return_code int out)
  end
end
```

The arguments passed are specified using the "in" parameter and the arguments to be returned are specified using the "out" parameter.

*callingConvention*

specifies that the arguments will be passed between functions and the calling code using the argument stack mechanism.

*dllName*

specifies the C shared library in which this function exists.

**Note:** The C shared library name can also be specified using the *vgj.defaultI4GLNativeLibrary* system property. If both *dllName* and the system property have been specified, the *dllName* will be used. For more information about the EGL nativeLibrary, see the *Library part of type nativeLibrary* help topic.

The C function receives an integer argument that specifies how many values were pushed on the argument stack (in this case, two arguments). This is the number of values to be popped off the stack in the C function. The function also needs to return values for the **msg\_status** and **return\_code** arguments before passing control back to the EGL program. The pop external functions are described in *Receiving values from EGL*; the return external functions are described in *Returning values to EGL*.

The C function should not assume it has been passed the correct number of stacked values. The C function should test its integer argument to see how many EGL arguments were stacked for it.

This example shows a C function that requires exactly one argument:

```
int nxt_bus_day(int nargs);
{
    int theDate;
    if (nargs != 1)
    {
        fprintf(stderr,
            "nxt_bus_day: wrong number of parms (%d)\n",
            nargs );
        ibm_lib4gl_returnDate(0L);
        return(1);
    }
    ibm_lib4gl_popDate(&theDate);
    switch(rdayofweek(theDate))
    {
        case 5: /* change friday -> monday */
            ++theDate;
        case 6: /* saturday -> monday*/
            ++theDate;
        default: /* (sun..thur) go to next day */
            ++theDate;
    }
    ibm_lib4gl_returnDate(theDate); /* stack result */
    return(1) /* return count of stacked */
}
```

The function returns the date of the next business day after a given date. Because the function must receive exactly one argument, the function checks for the number of arguments passed. If the function receives a different number of arguments, it terminates the program (with an identifying message).

**Related reference**

"BIGINT functions for C" on page 519



"C data types and EGL primitive types" on page 520  
 "Creating an EGL library part" on page 169  
 "DATE functions for C" on page 521  
 "DATETIME and INTERVAL functions for C" on page 522  
 "DECIMAL functions for C" on page 523  
 "Function invocations" on page 613  
 "Library part of type basicLibrary" on page 169  
 "Stack functions for C"  
 "Return functions for C" on page 528  
 "C functions with EGL" on page 516

## Stack functions for C

To call a C function, EGL uses an *argument stack*, a mechanism that passes arguments between the functions and the calling code. The EGL calling function pushes its arguments onto the stack and the called C function pops them off of the stack to use the values. The called function pushes its return values onto the stack and the caller pops them off to retrieve the values. The pop and return external functions are provided with the argument stack library. The pop external functions are described below according to the data type of the value that each pops from the argument stack. The return external functions are described in *Return functions for C*.

**Note:** The pop functions were originally used with IBM Informix 4GL (I4GL); hence the inclusion of "4gl" in the function names.

### Library functions for returning values

You can call the following library functions from a C function to pop number values from the argument stack:

- extern void ibm\_lib4gl\_popMInt(int \*iv)
- extern void ibm\_lib4gl\_popInt2(short \*siv)
- extern void ibm\_lib4gl\_popInt4(int \*liv)
- extern void ibm\_lib4gl\_popFloat(float \*fv)
- extern void ibm\_lib4gl\_popDouble(double \*dfv)
- extern void ibm\_lib4gl\_popDecimal(dec\_t \*decv)
- extern void ibm\_lib4gl\_popInt8(ifs\_int8\_t \*bi)

The following table and similar tables below map the return function names between I4GL pre-Version 7.31 and Version 7.31 and later:

Pre-Version 7.31 name	Version 7.31 and later name
popint	ibm_lib4gl_popMInt
popshort	ibm_lib4gl_popInt2
poplong	ibm_lib4gl_popInt4
popflo	ibm_lib4gl_popFloat
popdub	ibm_lib4gl_popDouble
popdec	ibm_lib4gl_popDecimal

Each of these functions, like all library functions for popping values, performs the following actions:

1. Removes one value from the argument stack.

2. Converts its data type if necessary. If the value on the stack cannot be converted to the specified type, an error occurs.
3. Copies the value to the designated variable.

The structure types **dec\_t** and **ifx\_int8\_t** are used to represent DECIMAL and BIGINT data in a C program. For more information about the **dec\_t** and **ifx\_int8\_t** structure types and library functions for manipulating and printing DECIMAL and BIGINT variables, see the *IBM Informix ESQL/C Programmer's Manual*.

### Library Functions for Popping Character Strings

You can call the following library functions to pop character values:

- `extern void ibm_lib4gl_popQuotedStr(char *qv, int len)`
- `extern void ibm_lib4gl_popString(char *qv, int len)`
- `extern void ibm_lib4gl_popVarChar(char *qv, int len)`

Pre-Version 7.31 name	Version 7.31 and later name
popquote	ibm_lib4gl_popQuotedStr
popstring	ibm_lib4gl_popString
popvchar	ibm_lib4gl_popVarChar

Both **ibm\_lib4gl\_popQuotedStr( )** and **ibm\_lib4gl\_popVarChar( )** copy exactly **len** bytes into the string buffer **\*qv**. Here **ibm\_lib4gl\_popQuotedStr( )** pads with spaces as necessary, but **ibm\_lib4gl\_popVarChar( )** does not pad to the full length. The final byte copied to the buffer is a null byte to terminate the string, so the maximum string data length is **len-1**. If the stacked argument is longer than **len-1**, its trailing bytes are lost.

The **len** argument sets the maximum size of the receiving string buffer. Using **ibm\_lib4gl\_popQuotedStr( )**, you receive exactly **len** bytes (including trailing blank spaces and the null), even if the value on the stack is an empty string. To find the true data length of a string retrieved by **ibm\_lib4gl\_popQuotedStr( )**, you must trim trailing spaces from the popped value.

**Note:** The functions **ibm\_lib4gl\_popString( )** and **ibm\_lib4gl\_popQuotedStr( )** are identical, except that **ibm\_lib4gl\_popString( )** automatically trims any trailing blanks.

### Library Functions for Popping Time Values

You can call the following library functions to pop DATE, INTERVAL, and DATETIME (TIMESTAMP) values:

- `extern void ibm_lib4gl_popDate(int *datv)`
- `extern void ibm_lib4gl_popInterval(intrvl_t *iv, int qual)`

You can call the following library function to pop TIMESTAMP values:

- `extern void ibm_lib4gl_popDateTime(dtime_t *dtv, int qual)`

Pre-Version 7.31 name	Version 7.31 and later name
popdate	ibm_lib4gl_popDate
popdtime	ibm_lib4gl_popDateTime

Pre-Version 7.31 name	Version 7.31 and later name
popinv	ibm_lib4gl_popInterval

The structure types **dtime\_t** and **intrvl\_t** are used to represent DATETIME and INTERVAL data in a C program. The **qual** argument receives the binary representation of the DATETIME or INTERVAL qualifier. For more information about the **dtime\_t** and **intrvl\_t** structure types and library functions for manipulating and printing DATE, DATETIME, and INTERVAL variables, see the *IBM Informix ESQL/C Programmer's Manual*.

### Library Functions for Popping BYTE or TEXT Values

You can call the following function to pop a BYTE or TEXT argument:

- `extern void ibm_lib4gl_popBlobLocator(loc_t **blob)`

Pre-Version 7.31 name	Version 7.31 and later name
poplocator	ibm_lib4gl_popBlobLocator

The structure type **loc\_t** defines a BYTE or TEXT value, and is discussed in the *IBM Informix ESQL/C Programmer's Manual*.

Any BYTE or TEXT argument must be popped as BYTE or TEXT because EGL provides no automatic data type conversion.

### Related reference

"BIGINT functions for C" on page 519  
 "C data types and EGL primitive types" on page 520  
 "C functions with EGL" on page 516  
 "DATE functions for C" on page 521  
 "DATETIME and INTERVAL functions for C" on page 522  
 "DECIMAL functions for C" on page 523  
 "Invoking a C Function from an EGL Program" on page 524  
 IBM Informix ESQL/C Programmer's Manual  
 "Return functions for C"

## Return functions for C

To call a C function, EGL uses an *argument stack*, a mechanism that passes arguments between the functions and the calling code. The EGL calling function pushes its arguments onto the stack and the called C function pops them off of the stack to use the values. The called function pushes its return values onto the stack and the caller pops them off to retrieve the values. The pop and return external functions are provided with the argument stack library. The return external functions are described below; the pop external functions used are described in *Stack functions for C*.

The external return functions copy their arguments to storage allocated outside the calling function. This storage is released when the returned value is popped. This situation makes it possible to return values from local variables of the function.

**Note:** The return functions were originally used with IBM Informix 4GL (I4GL); hence the inclusion of "4gl" in the function names.

### Library functions for returning values

The following library functions are available to return values:

- extern void `ibm_lib4gl_returnMInt(int iv)`
- extern void `ibm_lib4gl_returnInt2(short siv)`
- extern void `ibm_lib4gl_returnInt4(int lv)`
- extern void `ibm_lib4gl_returnFloat(float *fv)`
- extern void `ibm_lib4gl_returnDouble(double *dfv)`
- extern void `ibm_lib4gl_returnDecimal(dec_t *decv)`
- extern void `ibm_lib4gl_returnQuotedStr(char *str0)`
- extern void `ibm_lib4gl_returnString(char *str0)`
- extern void `ibm_lib4gl_returnVarChar(char *vc)`
- extern void `ibm_lib4gl_returnDate(int date)`
- extern void `ibm_lib4gl_returnDateTime(dtime_t *dtv)`
- extern void `ibm_lib4gl_returnInterval(intrvl_t *inv)`
- extern void `ibm_lib4gl_returnInt8(ifx_int8_t *bi)`

The following table maps the return function names between I4GL pre-Version 7.31 and Version 7.31 and later:

Pre-Version 7.31 name	Version 7.31 and later name
retint	<code>ibm_lib4gl_returnMInt</code>
retshort	<code>ibm_lib4gl_returnInt2</code>
retlong	<code>ibm_lib4gl_returnInt4</code>
retflo	<code>ibm_lib4gl_returnFloat</code>
retdub	<code>ibm_lib4gl_returnDouble</code>
retdec	<code>ibm_lib4gl_returnDecimal</code>
retquote	<code>ibm_lib4gl_returnQuotedStr</code>
retstring	<code>ibm_lib4gl_returnString</code>
retvchar	<code>ibm_lib4gl_returnVarChar</code>
retdate	<code>ibm_lib4gl_returnDate</code>
retmtime	<code>ibm_lib4gl_returnDateTime</code>
retinv	<code>ibm_lib4gl_returnInterval</code>

The argument of `ibm_lib4gl_returnQuotedStr( )` is a null-terminated string. The `ibm_lib4gl_returnString( )` function is included only for symmetry; it internally calls `ibm_lib4gl_returnQuotedStr( )`.

The C function can return data in whatever form is convenient. If conversion is possible, EGL converts the data type as required when popping the value. If data type conversion is not possible, an error occurs.

C functions called from EGL must always exit with the statement **return(*n*)**, where *n* is the number of return values pushed onto the stack. A function that returns nothing must exit with **return(0)**.

#### Related reference

“BIGINT functions for C” on page 519

“C data types and EGL primitive types” on page 520

“Invoking a C Function from an EGL Program” on page 524

“C functions with EGL” on page 516  
“DATE functions for C” on page 521  
“DATETIME and INTERVAL functions for C” on page 522  
“DECIMAL functions for C” on page 523  
“Stack functions for C” on page 526

---

## CICS-related considerations

This page provides miscellaneous information about running EGL-generated programs on CICS:

- “Record properties across programs”
- “Temporary storage queue access on CICS”

Also see the related links at the bottom of the page.

### Record properties across programs

In relation to EGL-generated programs that run on CICS for z/OS, the values of several record properties must be the same for each EGL record that accesses the same file in the same run unit. The record properties are as follows:

- File name
- Record type
- Format and length of the internal structure
- Length and offset of the key item, if any
- Offset of any variable length item in the record
- Offset of any number of occurs item in the record

### Temporary storage queue access on CICS

When a relative or serial record is associated with a temporary storage queue on CICS, EGL adds a *deletion byte* to the beginning of the EGL record associated with the queue. The deletion byte is not in the record definition itself, but non-EGL programs that share the temporary storage queue must allocate space for the byte, which has either of two values:

- One (1) means that the record was deleted and that the record length is 1
- Zero (0) means that the record exists logically in the file

EGL statements that operate on a temporary storage queue act as follows:

- The add or replace statement sets the deletion byte to 0
- The delete statement sets the deletion byte and record length to 1
- If the deletion byte equals 1, the inquiry or update statement sets the EGL error value NRF
- The get next statement skips each queue record in which the deletion byte is 1

#### Related concepts

“Record types and properties” on page 138  
“Record parts” on page 135

#### Related tasks

“Setting up the J2EE server for CICSJ2C calls” on page 441

### Related reference

“add” on page 661

“get” on page 687

“get next” on page 701

“delete” on page 673

“I/O error values” on page 638

“Record and file type cross-reference” on page 860

“replace” on page 738

“terminalID” on page 1075

---

## Comments

A *comment* in an EGL file is created in either of the following ways:

- Double right slashes (//) indicate that the subsequent characters are a comment, up to and including the end-of-line character
- A single or multiline comment is delimited by a right slash and asterisk at the start (/\*) and by an asterisk and right slash at the end (\*/); this form of comment is valid anywhere that a white-space character is valid

You may place a comment inside or outside of an executable statement, as in this example:

```
/* the assignment e = f occurs if a == b or if c == d */
if (a == b           // one comparison
    || /* OR; another comparison */ c == d)
    e = f;
end
```

EGL does not support embedded comments, so the following entries cause an error:

```
/* this line starts a comment /* and
   this line ends the comment, */
   but this line is not inside a comment at all */
```

The comment in the first two lines includes a second comment delimiter (/\*). An error results only when EGL tries to interpret the third line as source code.

The following is valid:

```
a = b;   /* this line starts a comment // and
          this line ends the comment */
```

The double right slashes (//) in the last example are themselves part of a larger comment.

Between the symbols #sql{ and }, the EGL comments described earlier are not valid. The following statements apply:

- An SQL comment begins with a double hyphen (--) at the beginning of a line or after white space and continues until the end of the line
- Comments are not available inside a string literal. A series of characters in that literal is interpreted as text even in these contexts:
  - A prepare statement
  - The **defaultSelectCondition** property of a record of type SQLRecord

### Related concepts

“EGL projects, packages, and files” on page 15

#### Related reference

“EGL source format” on page 586

“EGL statements” on page 88

---

## Compatibility with VisualAge Generator

EGL is the replacement for VisualAge Generator 4.5 and includes some syntax primarily to enable the migration of existing programs to the new development environment. This syntax is supported in the development environment if the EGL preference **VAGCompatibility** is selected or (at generation or debug time) if the build descriptor option **VAGCompatibility** is set to *yes*. The setting of the preference also establishes the default value of the build descriptor option.

The following statements apply when VisualAge Generator compatibility is in effect:

- Three otherwise invalid characters (- @ #) are valid in identifiers, although each is invalid as an initial character in any case; for details, see *Naming conventions*
- If you refer to a static, single-dimension array of structure items without specifying an index, the array index defaults to 1; for details, see *Arrays*
- The primitive types NUMC and PACF are available, as described in *Primitive types*
- If you specify an even length for an item of primitive type DECIMAL, EGL increments the length by one except when the item is used as an SQL host variable.
- The SQL item property **SQLDataCode** is available, as described in *SQL item properties*
- A set of call options are available in the call statement
- The option **externallyDefined** is in the statements **show** and **transfer**
- The following system variables are available:
  - **VGVar.handleSysLibraryErrors**
  - **ConverseVar.segmentedMode**
- The following system functions are available:
  - **VGLib.getVAGSysType**
  - **VGLib.connectionService**
- You can issue a statement of the following form:

`display printForm`

*printForm*

Name of a print form that is visible to the program.

In that case, **display** is equivalent to **print**.

- The following program properties are available in all cases and are especially useful for code that was written in VisualAge Generator:
  - **allowUnqualifiedItemReferences**
  - **handleHardIOErrors** (when set to *no*)
  - **includeReferencedFunctions**
  - **localSQLScope** (when set to *yes*)
  - **throwNrfEofExceptions** (when set to *yes*)

For details, see *Program part properties*.

In relation to DL/I code, the @DLI program property, field **handleHardDLIErrors** is useful for code that was written in VisualAge Generator, if that property is set to *no*. For details, see *@DLI*.

- If you set the text-form property **value**, the content of that property is available in the program only after the user has returned the form. For this reason, the value that you set in the program does not need to be valid for the item in the program.

For access to full details on migrating VisualAge Generator programs to EGL, see *Sources of additional information on EGL*.

#### Related concepts

"Sources of additional information on EGL" on page 14

#### Related reference

"@DLI" on page 322

"Arrays" on page 75

"call" on page 665

"Input form" on page 859

"Input record" on page 859

"Naming conventions" on page 778

"pfKeyEquate" on page 792

"Primitive types" on page 34

"print" on page 737

"Program part in EGL source format" on page 841

"Program part properties" on page 856

"show" on page 751

"SQL item properties" on page 68

"connectionService()" on page 1047

"getVAGSysType()" on page 1054

"handleSysLibraryErrors" on page 1084

"segmentedMode" on page 1061

"transfer" on page 752

---

## ConsoleUI

### ConsoleField properties and fields

The following properties are required in a variable of type ConsoleField:

- **fieldLen** (unless the ConsoleField is a constant field)
- **position**

The **name** field is also required, although not in a constant ConsoleField.

The properties of ConsoleField are as follows:

#### fieldLen

Specifies the number of positions needed to display the largest value of interest. For constant consoleFields, you do not set this property: **fieldLen** is the number of characters occupied by the displayed value, as included in the **value** property.

**Type:** *INT*

**Example:** *fieldLen = 20*

**Default:** *none*



### **position**

The location of the console field within the form. The property contains an array of two positive integers: the line number followed by the column number. The line number is calculated from the top of the form. Similarly, the column number is calculated from the left of the form.

**Type:** *INT[]*

**Example:** *position = [2, 3]*

**Default:** *[1,1]*

### **segments**

Specifies the row, column, and length of each *field segment*, which is a `consoleField` subsection that can have delimiters. To create the appearance of a multiline text box, you stack one field segment on successive lines at the same form column, and the collection of segments acts as one field.

**Type:** *INT[3][]*

**Example:** *segments = [[5,1,10],[6,1,10]]*

**Default:** *none*

If a value is specified for **segments**, the value for **position** is ignored, and **fieldLen** should be set to the length of all segments combined.

If you specify multiple segments, the behavior of the `ConsoleField` is also affected by the value of the **lineWrap** field.

### **validValues**

Specifies the list of values that are valid for user input.

**Type:** *Array literal of singular and two-value elements*

**Example:** *validValues = [ [1,3], 5, 12 ]*

**Default:** *none*

For details, see *validValues*.

The properties of a `consoleField` array include the previous ones (except for **segments**), as well as these:

### **columns**

Specifies the number of columns in which to display the elements in an array of type `ConsoleField`. If the array has five elements and the value of the **columns** property is two, for example, the first line of the form shows two elements; the second line shows two elements; and the third line shows one element.

**Type:** *INT*

**Example:** *columns = 3*

**Default:** *1*

This property is meaningful only for arrays of type `ConsoleField`. The distribution of array elements on screen (whether across or up and down) is affected by the property **orientIndexAcross**.

### **linesBetweenRows**

Specifies the number of blank lines between each line that contains an array element.

**Type:** *INT*

**Example:** *linesBetweenRows = 3*

**Default:** *0*

This property is meaningful only for arrays of type `ConsoleField`.

**orientIndexAcross**

Indicates whether the distribution of array elements is across the screen, as shown in a later example.

**Type:** *Boolean*

**Example:** *orientIndexAcross = yes*

**Default:** *yes*

This property is meaningful only for arrays of type `consoleField`.

If the property **orientIndexAcross** is set to *yes*, successive elements of the array are displayed from left to right. In the following, 2-column example, each successive element displays an integer that is equivalent to the element index:

```
1  2
3  4
5
```

If the property **orientIndexAcross** is set to *no*, the successive elements are displayed from top to bottom:

```
1  4
2  5
3
```

**spacesBetweenColumns**

Specifies the number of spaces separating each column of fields.

**Type:** *INT*

**Example:** *spacesBetweenColumns = 3*

**Default:** *1*

This property is valid only for arrays of type `consoleField`.

The fields of `ConsoleField` are as follows:

**align**

The **align** field specifies the position of data in a variable field when the length of the data is smaller than the length of the field.

**Type:** *AlignKind*

**Example:** *align = left*

**Default:** *left for character or timestamp data, right for numeric*

**Updatable at run time?** *Yes*

Values are as follows:

**left**

Place the data at the left of the field. Initial spaces are stripped and placed at the end of the field.

**none**

Do not justify the data. This setting is valid only for character data.

**right**

Place the data at the right of the field. Trailing spaces are stripped and placed at the beginning of the field.

**autonext**

Indicates whether, after the user fills the current `ConsoleField`, the cursor goes to the next field.

**Type:** *Boolean*

**Example:** *autonext = yes*

**Default:** *None*

**Updatable at run time?** *Yes*

The tab order determines which ConsoleField is next, as described in *ConsoleUI parts and related variables*.

### **binding**

Specifies the name of the variable to which the ConsoleField is bound by default.

**Type:** *String*

**Example:** *binding = "myVar"*

**Default:** *None*

**Updatable at run time?** *No.*

For an overview of binding, see *ConsoleUI parts and related variables*.

### **caseFormat**

Specifies how to treat input and output in relation to case sensitivity.

**Type:** *CaseFormatKind*

**Example:** *caseFormat = lowerCase*

**Default:** *defaultCase*

**Updatable at run time?** *Yes*

Values are as follows:

#### **defaultCase (the default)**

Has no effect on case

#### **lowerCase**

Transforms characters to lowercase, as possible

#### **upperCase**

Transforms characters to uppercase, as possible

### **color**

Specifies the color of the text in the ConsoleField.

**Type:** *ColorKind*

**Example:** *color = red*

**Default:** *white*

**Updatable at run time?** *Yes, but the update has a visual effect only if the ConsoleField is displayed (or obtains focus) after the field is updated*

Values are as follows:

#### **defaultColor or white (the default)**

White

#### **black**

Black

#### **blue**

Blue

#### **cyan**

Cyan

#### **green**

Green

#### **magenta**

Magenta

**red**

Red

**yellow**

Yellow

**comment**

Specifies the *comment*, which is the text displayed in the Window-specific comment line (if any) when the cursor is in the ConsoleField.

**Type:** *String*

**Example:** *"Employee name"*

**Default:** *Empty string*

**Updatable at run time?** *No*

**commentKey**

Specifies a key used to search the resource bundle that includes the *comment*, which is the text displayed in the Window-specific comment line (if any) when the cursor is in the ConsoleField. If you specify both **comment** and **commentKey**, **comment** is used.

**Type:** *String*

**Example:** *commentKey = "myKey"*

**Default:** *Empty string*

**Updatable at run time?** *No*

The resource bundle is identified by the system variable

**ConsoleLib.messageResource**, as described in *messageResource*.

**dataType**

Specifies a string to identify a data type. The value is used to validate that user input (such as = 1.5) is compatible with a particular kind of SQL column. The field is meaningful only when the **openUI** statement for the ConsoleField (or related ConsoleForm) includes the statement property **isConstruct**.

**Type:** *String*

**Example:** *dataType = "NUMBER"*

**Default:** *Empty string*

**Updatable at run time?** *No*

In relation to numeric input, specify the value *"NUMBER"* if you allow the user to specify a floating point value (in which case, > 1.5 is valid user input); otherwise, specify the string equivalent of an integer; for example, *"INT"*.

**dateFormat**

Indicates how to format output; but specify **dateFormat** only if the ConsoleField accepts a date.

**Type:** *a String or date-related system constant*

**Example:** *dateFormat = isoDateFormat*

**Default:** *none*

**Updatable at run time?** *No*

Valid values are as follows:

*"pattern"*

The value of *pattern* consists of a set of characters, as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete date specification, but not from the middle.

**defaultDateFormat**

The date format given in the runtime Java locale.

**isoDateFormat**

The pattern "yyyy-MM-dd", which is the date format specified by the International Standards Organization (ISO).

**usaDateFormat**

The pattern "MM/dd/yyyy", which is the IBM USA standard date format.

**eurDateFormat**

The pattern "dd.MM.yyyy", which is the IBM European standard date format.

**jisDateFormat**

The pattern "yyyy-MM-dd", which is the Japanese Industrial Standard date format.

**systemGregorianCalendarFormat**

An 8- or 10-character pattern that includes dd (for numeric day of the month), MM (for numeric month), and yy or yyyy (for numeric year), with characters other than d, M, y, or digits used as separators.

The format is in this Java runtime property:

```
vgj.datemask.gregorian.long.NLS
```

*NLS*

The NLS (national language support) code that is specified in the Java runtime property **vgj.nls.code**. The code is one of those listed in targetNLS. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java runtime properties (details)*.

**systemJulianDateFormat**

A 6- or 8-character pattern that includes DDD (for numeric day of the year) and yy or yyyy (for numeric year), with characters other than D, y, or digits used as separators.

The format is in this Java runtime property:

```
vgj.datemask.julian.long.NLS
```

*NLS*

The NLS (national language support) code that is specified in the Java runtime property **vgj.nls.code**. The code is one of those listed in targetNLS. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java runtime properties (details)*.

**editor**

Specifies the program for user interaction with the data; but is meaningful only if the ConsoleField is bound to a variable of type LOB.

**Type:** *String*

**Example:** *editor = "/bin/vi"*

**Default:** *none*

**Updatable at run time?** *Yes*

You can specify the name of an executable found in the PATH or LIBPATH; alternatively, you can specify the fully qualified path of that executable.

### **help**

Specifies the text to display when the following situation is in effect:

- The cursor is in the ConsoleField; and
- The user presses the key identified in **ConsoleLib.key\_help**.

**Type:** *String*

**Example:** *help = "Update the value"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

### **helpKey**

Specifies an access key for searching the resource bundle that contains text for display when the following situation is in effect:

- The cursor is in the ConsoleField; and
- The user presses the key identified in **ConsoleLib.key\_help**.

If you specify both **help** and **helpKey**, **help** is used.

**Type:** *String*

**Example:** *helpKey = "myKey"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

### **highlight**

Specifies the special effects (if any) that are used when displaying the ConsoleField.

**Type:** *HighlightKind[]*

**Example:** *highlight = [reverse, underline]*

**Default:** *[noHighLight]*

**Updatable at run time?** *Yes, but the update has a visual effect only if the ConsoleField is displayed (or obtains focus) after the **highlight** field is updated*

Values are as follows:

#### **noHighlight (the default)**

Causes no special effect. Use of this value overrides any other.

#### **blink**

Has no effect.

#### **reverse**

Reverses the text and background colors so that (for example) if the display has a black background with white letters, the background becomes white and the text becomes black.

#### **underline**

Places an underline under the affected areas. The color of the underline is the color of the text, even if the value **reverse** is also specified.

### **initialValue**

Specifies the initial value for display. If you specify both **initialValue** and **initialValueKey**, **initialValue** is used.

**Type:** *String*

**Example:** *initialValue = "200"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

If the **setInitial** property in the **openUI** statement is set to true, the value of the **initialValue** property in the **consoleField** is used. If that **openUI** property is false, however, current values of bound variables are shown instead, and the value of the **initialValue** property is ignored.

#### **initialValueKey**

Specifies an access key for searching the resource bundle that contains the initial value for display. If you specify both **initialValue** and **initialValueKey**, **initialValue** is used.

**Type:** *String*

**Example:** *initialValueKey = "myKey"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

#### **inputRequired**

Indicates whether the user will be prevented from navigating away from the field without entering a value.

**Type:** *Boolean*

**Example:** *inputRequired = yes*

**Default:** *no*

**Updatable at run time?** *No*

#### **intensity**

Specifies the strength of the displayed font.

**Type:** *IntensityKind[]*

**Example:** *intensity = [bold]*

**Default:** *[normalIntensity]*

**Updatable at run time?** *Yes, but the update has a visual effect only if the ConsoleField is displayed (or obtains focus) after the **intensity** field is updated*

Values are as follows:

##### **normalIntensity (the default)**

Causes no special effect. Use of this value overrides any other.

##### **bold**

Causes the text to appear in a bold-weight font.

##### **dim**

Has no effect at this time. In future, may cause the text to appear with a lessened intensity, as appropriate when an input field is either disabled or should be deemphasized.

##### **invisible**

Removes any indication that the field is on the form.

#### **isBoolean**

Indicates whether the **ConsoleField** represents a Boolean value. The field **isBoolean** restricts the valid **ConsoleField** values and is useful for input or output.

The value of a numeric field is 0 (for false) or 1 (for true).

The value of a character field is represented by a word or subset of a word that is national-language dependent, and the specific values are determined by the locale. In English, for example, a boolean field of three or more characters has the value *yes* (for true) or *no* (for false), and a one-character boolean field value has the truncated value *y* or *n*.

**Type:** *Boolean*

**Example:** *isBoolean = yes*

**Default:** *no*

**Updatable at run time?** *No*

### **lineWrap**

Indicates how to wrap text onto a new line whenever wrapping is necessary to avoid truncating text.

**Type:** *LineWrapType*

**Example:** *value = compress*

**Default:** *character*

**Updatable at run time?** *Yes*

Values are as follows:

#### **character (the default)**

The text in a field will not be split at a white space, but at the character position where the boundary of the field segment is.

#### **compress**

If possible, the text will be split at a white space. When the user leaves the consoleField (either by navigating to another consoleField or by pressing **Esc**), the value is assigned to the bound variable, and any additional spaces that are used to wrap text are removed.

#### **word**

If possible, the text in a field will be split at a white space. When the value is assigned to the bound variable, additional spaces are included to reflect how the value was padded to wrap at word boundaries.

The **lineWrap** field is meaningful only for a ConsoleField that has multiple segments, as is controlled by the **segments** property.

### **masked**

Indicates whether each character in the ConsoleField is displayed as an asterisk (\*), as is appropriate when the user types a password.

**Type:** *Boolean*

**Example:** *masked = yes*

**Default:** *no*

**Updatable at run time?** *Yes*

### **minimumInput**

Indicates the minimum number of characters in valid input.

**Type:** *INT*

**Example:** *minimumInput = 4*

**Default:** *no*

**Updatable at run time?** *No*



**name**

ConsoleField name, as used in a programming context in which the name is resolved at run time. It is strongly recommended that the value of the name field be the same as the name of the variable.

**Type:** *String*

**Example:** *name = "myField"*

**Default:** *none*

**Updatable at run time?** *No*

**numericFormat**

Indicates how to format output; but specify **numericFormat** only if the ConsoleField accepts a number.

**Type:** *String*

**Example:** *numericFormat = "-###@"*

**Default:** *none*

**Updatable at run time?** *No*

Valid characters are as follows:

- # A placeholder for a digit.
- \* Use an asterisk (\*) as the fill character for a leading zero.
- & Use a zero as the fill character for a leading zero.
- # Use a space as the fill character for a leading zero.
- < Left justify the number.
- , Use a locale-dependent numeric separator unless the position contains a leading zero.
- . Use a locale-dependent decimal point.
- Use a minus sign (-) for values less than 0; use a space for values greater than or equal to 0.
- + Use a minus sign for values less than 0; use a plus sign (+) for values greater than or equal to 0.
- ( Precede negative values with a left parenthesis, as appropriate in accounting.
- ) Place a right parenthesis after a negative value, as appropriate in accounting.
- \$ Precede the value with the locale-dependent currency symbol.
- @ Place the locale-dependent currency symbol after the value.

**pattern**

Specifies the pattern for input and output formatting if the ConsoleField content is of a character type.

**Type:** *String*

**Example:** *pattern = "(###) ###-####"*

**Default:** *none*

**Updatable at run time?** *No*

These control characters are available:

- *A* is a placeholder for letters, and the subset of characters that are considered to be letters is dependent on the locale
- *#* is a placeholder for numeric digits
- *X* is a placeholder for a required character of any kind

Characters other than the preceding three are included in the input or output; but for output, any overlaid characters are lost:

- If the output pattern is "(###) ###-####", the value "6219655561212" is shown as follows:

(219) 555-1212

Each 6 in the original value is unavailable to the user and is lost if the data store is updated.

- For input, the cursor skips the literal characters and only allows typing where the placeholder characters occur. In the current example, if the user types 2195551212, the string "(219) 555-1212" becomes the value within the ConsoleField and is the value placed in the bound variable.

### **protect**

Specifies whether the ConsoleField is protected from user update.

**Type:** *Boolean*

**Example:** *protect = yes*

**Default:** *no*

**Updatable at run time?** *No*

Values are as follows:

#### **No (the default)**

Sets the field so that the user can overwrite the value in it.

#### **Yes**

Sets the consoleField so that the user cannot overwrite the value in it. In addition, the cursor skips the consoleField whenever the user attempts to navigate to it, as in these cases:

- The user is working on the previous consoleField in the tab order and either (a) presses **Tab** or (b) fills that previous consoleField with content when field **autonext** is set to yes.
- The user is working on the next consoleField in the tab order and presses **Shift Tab**.
- The user uses arrow keys to move to the next or previous consoleField.

You can bind a variable to a consoleField that is protected or not. The setting of the openUI property **setInitial** determines whether the value of the bound variable is displayed.

A runtime error occurs if the program tries to move to a consoleField that is protected.

### **SQLColumnName**

Specifies the name of the database table column that is associated with the ConsoleField. The name is used to create search criteria when the **openUI** statement for the ConsoleField (or related ConsoleForm) includes the statement property **isConstruct**.

**Type:** *String*

**Example:** *SQLColumnName = "ID"*

**Default:** *none*

**Updatable at run time?** *Yes*

#### **timeFormat**

Indicates how to format output; but specify **timeFormat** only if the ConsoleField accepts a time.

**Type:** *a String or time-related system constant*

**Example:** *timeFormat = isoTimeFormat*

**Default:** *none*

**Updatable at run time?** *No*

Valid values are as follows:

*"pattern"*

The value of *pattern* consists of a set of characters, as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete time specification, but not from the middle.

#### **defaultTimeFormat**

The time format given in the runtime Java locale.

#### **isoTimeFormat**

The pattern "HH.mm.ss", which is the time format specified by the International Standards Organization (ISO).

#### **usaTimeFormat**

The pattern "hh:mm AM", which is the IBM USA standard time format.

#### **eurTimeFormat**

The pattern "HH.mm.ss", which is the IBM European standard time format.

#### **jisTimeFormat**

The pattern "HH:mm:ss", which is the Japanese Industrial Standard time format.

#### **timestampFormat**

Indicates how to format output; but specify **timestampFormat** only if the ConsoleField accepts a timestamp.

**Type:** *a String or timestamp-related system constant*

**Example:** *timestampFormat = odbcTimestampFormat*

**Default:** *none*

**Updatable at run time?** *No*

Valid values are as follows:

*"pattern"*

The value of *pattern* consists of a set of characters, as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete timestamp specification, but not from the middle.

#### **defaultTimestampFormat**

The timestamp format given in the runtime Java locale.

**db2TimestampFormat**

The pattern "yyyy-MM-dd-HH.mm.ss.ffffff", which is the IBM DB2 default timestamp format.

**odbcTimestampFormat**

The pattern "yyyy-MM-dd HH:mm:ss.ffffff", which is the ODBC timestamp format.

**value**

The current value displayed in the consoleField. Your code can set this value so that invocation of **ConsoleLib.displayForm** displays the specified value in the consoleField.

**Type:** *String*

**Example:** *value = "View"*

**Default:** *none*

**Updatable at run time?** *Yes*

**verify**

Indicates whether the user is prompted to retype the same value after trying to exit the ConsoleField.

**Type:** *Boolean*

**Example:** *verify = yes*

**Default:** *no*

**Updatable at run time?** *No*

Values are as follows:

**No (the default)**

EGL run time does not issue a special prompt.

**Yes**

When the user tries to leave the ConsoleField, the EGL runtime acts as follows:

- Clears the consoleField, keeping the cursor there
- Displays a message for the user to repeat the entry
- Compares the two input values when the user tries to leave the consoleField again

If the values match, the bound variable receives that value and processing continues as usual. If the values do not match, the consoleField content reverts to the value that preceded the first of the two user inputs, and the cursor remains in the field.

**Related concepts**

"Console user interface" on page 207

**Related reference**

"ConsoleUI parts and related variables" on page 209

"Date, time, and timestamp format specifiers" on page 46

"Java runtime properties (details)" on page 642 "openUI" on page 726

"validValues" on page 834

**Related task**

"Creating an interface with ConsoleUI" on page 208

## ConsoleForm properties in EGL consoleUI

The properties of a record part of type ConsoleForm are as follows, and only formSize is required:

### delimiters

Specifies the characters that are displayed before and after input fields. The characters are displayed only if the value of property **showBrackets** is *yes*.

**Type:** *String literal*

**Example:** *delimiters = "<>/"*

**Default:** *"[]|"*

Wherever possible, the first character is displayed before each non-constant ConsoleField, and the second character is displayed after each non-constant ConsoleField. However, the third character is displayed between two non-constant ConsoleFields that are separated by a single position.

If you specify fewer than three characters, a default character is in effect for each unspecified character. If you specify more than three characters, the fourth and subsequent characters are ignored.

### formSize

The dimensions of the form. The field must contain an array of two positive integers: the number of lines followed by the number of columns.

**Type:** *INT[2]*

**Example:** *formSize = [24, 80]*

**Default:** *none*

If either dimension exceeds the size of the window in which the form is displayed, the form size is reduced to fit the window dimensions. However, if a ConsoleField cannot fit into the window, the program ends.

### name

Form name, as used in a programming context in which the name is resolved at run time. It is recommended that the value of the name field, if any, be the same as the name of the variable.

**Type:** *String*

**Example:** *name = "myForm"*

**Default:** *none*

The name field is used in system functions such as **ConsoleLib.displayFormByName**.

### showBrackets

Indicates whether the non-constant ConsoleFields are delimited by a pair of characters such as brackets.

**Type:** *Boolean*

**Example:** *showBrackets = no*

**Default:** *yes*

For other details, see the property **delimiters**.

### Related concepts

"Console user interface" on page 207

### Related reference

"ConsoleUI parts and related variables" on page 209

"openUI" on page 726

### Related task

“Creating an interface with ConsoleUI” on page 208

## Menu fields in EGL consoleUI

The following list defines the fields in a variable of type `Menu`. You must specify the field **labelText** or **labelKey**.

### labelText

The label that is displayed to the left of the list of menuItems.

**Type:** *String literal*

**Example:** *labelText = "Options: ".*

**Default:** *none.*

**Updatable at run time?** *No*

### labelKey

Specifies a key for searching the resource bundle that contains the menu label. If you specify both **labelText** and **labelKey**, **labelText** is used.

**Type:** *String*

**Example:** *labelKey = "myKey"*

**Default:** *Empty string*

**Updatable at run time?** *No*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

### menuItems

An array of menu items, each of which is declared in the program or is created dynamically with the keyword **new**. For details on the second option, see *Use of new in consoleUI*.

**Type:** *MenuItem[]*

**Example:** *menuItems = [myItem, new MenuItem {name = "Remove", labelText = "Delete all"}].*

**Default:** *none.*

**Updatable at run time?** *No*

You can add a menuItem in the program by using the following syntax:

```
myMenu.menuItems.addElement(myMenuItem)
```

*myMenu*

Name of the variable of type `Menu`.

*myMenuItem*

Name of the variable of type `MenuItem`.

The program ends if you issue an **openUI** statement for a menu on which no menuItems exist.

### Related concepts

“Console user interface” on page 207

### Related reference

“Arrays” on page 75

“ConsoleUI parts and related variables” on page 209

“openUI” on page 726  
“MenuItem fields in EGL consoleUI”  
“Use of new in ConsoleUI” on page 212

**Related task**

“Creating an interface with ConsoleUI” on page 208

## MenuItem fields in EGL consoleUI

The following list defines the consoleFields in a variable of type MenuItem. None of the consoleFields is required; you can determine the user’s selection by setting any of three fields: **accelerators**, **labelText**, or **labelKey**.

**accelerators**

Indicates keystrokes that are equivalent to the user’s selection of the menuItem. Each of those keystrokes causes execution of the **openUI** statement’s OnEvent clause that corresponds to the menuItem selection.

**Type:** *String[]*

**Example:** *accelerators = ["F1", "ALT\_F1"]*

**Default:** *none*

**Updatable at run time?** *No*

**comment**

Specifies the *comment*, which is the text displayed in the menuItem-specific comment line when the menuItem is selected.

**Type:** *String*

**Example:** *"Delete the record"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

The comment line is one beneath the menu line.

**commentKey**

Specifies a key used to search the resource bundle that includes the *comment*, which is text displayed in the menuItem-specific comment line (if any) when the menuItem is selected. If you specify both **comment** and **commentKey**, **comment** is used.

**Type:** *String*

**Example:** *commentKey = "myKey"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

**help**

Specifies the text to display when the following situation is in effect:

- The menuItem is selected; and
- The user presses the key identified in **ConsoleLib.key\_help**.

**Type:** *String*

**Example:** *help = "Deletion is permanent"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

### **helpKey**

Specifies an access key for searching the resource bundle that contains text for display when the following situation is in effect:

- The menuItem is selected; and
- The user presses the key identified in **ConsoleLib.key\_help**.

If you specify both **help** and **helpKey**, **help** is used.

**Type:** *String*

**Example:** *helpKey = "myKey"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

### **labelText**

The label that represents the menuItem.

**Type:** *String literal*

**Example:** *labelText = "Delete".*

**Default:** *none.*

**Updatable at run time?** *No*

### **labelKey**

Specifies a key for searching the resource bundle that contains the menuItem label. If you specify both **labelText** and **labelKey**, **labelText** is used.

**Type:** *String*

**Example:** *labelKey = "myKey"*

**Default:** *Empty string*

**Updatable at run time?** *No*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

### **name**

MenuItem name, as used in a programming context in which the name is resolved at run time. In particular, the name is used in the **openUI** statement that responds to the menuItem selection.

It is recommended that the value of the name field be the same as the name of the variable.

**Type:** *String*

**Example:** *name = "myItem"*

**Default:** *none*

**Updatable at run time?** *No*

### **Related concepts**

"Console user interface" on page 207

### **Related reference**

"ConsoleUI parts and related variables" on page 209

"Menu fields in EGL consoleUI" on page 547

"openUI" on page 726



### Related task

“Creating an interface with ConsoleUI” on page 208

## PresentationAttributes fields in EGL consoleUI

The following list defines the fields that you can set or retrieve in any system variable of type PresentationAttributes:

### color

Specifies a color.

**Type:** *ColorKind*

**Example:** *color = red*

**Default:** *white*

**Updatable at run time?** *Yes, but the update has a visual effect only for output that is displayed after the color field is updated*

Values are as follows:

#### defaultColor or white (the default)

White

#### black

Black

#### blue

Blue

#### cyan

Cyan

#### green

Green

#### magenta

Magenta

#### red

Red

#### yellow

Yellow

### highlight

Specifies the special effects (if any) that are used when displaying output.

**Type:** *HighlightKind[]*

**Example:** *highlight = [reverse, underline]*

**Default:** *[noHighLight]*

**Updatable at run time?** *Yes, but the update has a visual effect only for output that is displayed after the highlight field is updated*

Values are as follows:

#### noHighlight (the default)

Causes no special effect. Use of this value overrides any other.

#### blink

Has no effect at this time.

#### reverse

Reverses the text and background colors so that (for example) if the

display has a black background with white letters, the background becomes white and the text becomes black.

**underline**

Places an underline under the affected areas. The color of the underline is the color of the text, even if the value **reverse** is also specified.

**intensity**

Specifies the strength of the displayed font.

**Type:** *IntensityKind[]*

**Example:** *intensity = [bold]*

**Default:** *[normalIntensity]*

**Updatable at run time?** *Yes, but the update has a visual effect only for output that is displayed after the intensity field is updated*

Values are as follows:

**normalIntensity (the default)**

Causes no special effect. Use of this value overrides any other.

**bold**

Causes the text to appear in a bold-weight font.

**dim**

Has no effect at this time. In future, may cause the text to appear with a lessened intensity, as appropriate when all input fields are disabled.

**invisible**

Removes any indication that the text is on the form.

**Related concepts**

"Console user interface" on page 207

**Related reference**

"currentDisplayAttrs" on page 897

"currentRowAttrs" on page 897

"defaultDisplayAttributes" on page 898

"defaultInputAttributes" on page 898

"ConsoleUI parts and related variables" on page 209

"openUI" on page 726

**Related task**

"Creating an interface with ConsoleUI" on page 208

## Prompt fields in EGL consoleUI

The following list defines the fields in a variable of type Prompt. None of the fields is required.

**isChar**

Indicates whether, after the prompt is displayed, the user's first keystroke ends the operation.

**Type:** *Boolean*

**Example:** *isChar = yes*

**Default:** *no*

**Updatable at run time?** *Yes*

Values are as follows:

**no (the default)**

The operation ends when the user presses **Enter** or presses a key associated with an OnEvent clause of the **openUI** statement that displays the prompt. The variable to which the prompt is bound receives the input characters.

**yes**

The user's first keystroke ends the operation. The variable to which the prompt is bound receives the character, if the character is printable.

In either case, you can respond to a particular keystroke by setting an OnEvent clause of type ON\_KEY.

**message**

Specifies the text that prompts the user.

**Type:** *String*

**Example:** *message = "Type here: "*

**Default:** *Empty string*

**Updatable at run time?** *Yes, before your code issues the **openUI** statement*

**messageKey**

Specifies a key used to search the resource bundle that includes the prompt text. If you specify both **message** and **messageKey**, **message** is used.

**Type:** *String*

**Example:** *messageKey = "promptText"*

**Default:** *Empty string*

**Updatable at run time?** *Yes*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

**responseAttr**

Specifies the presentation attributes that are used when displaying user input.

**Type:** *PresentationAttributes literal*

**Example:** *responseAttr {color = green, highlight = [underline], intensity = [bold]}*

**Default:** *no*

**Updatable at run time?** *Yes*

This field has an effect only if the field **isChar** is set to *no*.

For details on **responseAttr** values, see *PresentationAttributes fields in EGL consoleUI*.

**Related concepts**

"Console user interface" on page 207

**Related reference**

"ConsoleUI parts and related variables" on page 209

"Java runtime properties (details)" on page 642

"messageResource" on page 910

"openUI" on page 726

"PresentationAttributes fields in EGL consoleUI" on page 550

**Related task**

"Creating an interface with ConsoleUI" on page 208

## Window fields in EGL consoleUI

The following list defines the fields in a variable of type Window. None of the fields is required, but **size** is needed in practice.

### color

Specifies the color that is used when displaying the following kinds of output in the window:

- Labels in consoleForms
- Input fields in prompts
- Window border
- Output of system functions such as **ConsoleLib.displayAtPosition**

**Type:** *ColorKind*

**Example:** *color = red*

**Default:** *white*

**Updatable at run time?** *Yes, but the update has a visual effect only if you open the window after the field is updated*

Values are as follows:

#### **defaultColor or white (the default)**

White

#### **black**

Black

#### **blue**

Blue

#### **cyan**

Cyan

#### **green**

Green

#### **magenta**

Magenta

#### **red**

Red

#### **yellow**

Yellow

### commentLine

Sets the number of the line at which a comment (if any) is displayed if the Window field **hasCommentLine** is set to *yes*. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on).

**Type:** *INT*

**Example:** *commentLine = 10*

**Default:** *Last line of the window (although if only the screen window is open, the comment is on the second to last line of that window)*

**Updatable at run time?** *Yes, but the update has a visual effect only if you open the window after the field is updated*

The validity of the value is determined only at run time.

**formLine**

Sets the number of the line at which forms are displayed. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on).

**Type:** *INT*

**Example:** *formLine = 8*

**Default:** *3*

**Updatable at run time?** *Yes, but the update has a visual effect only if the window is displayed after the field is updated*

The validity of the value is determined only at run time.

**hasBorder**

Indicates whether the window is surrounded by a border. If the value is *yes*, the color of the border is specified in the Window field *color*.

**Type:** *Boolean*

**Example:** *hasBorder = yes*

**Default:** *no*

**Updatable at run time?** *Yes, but the update has a visual effect only if you open the window after the field is updated*

**hasCommentLine**

Indicates whether the window reserves a line for *comments*, which are text entries that are displayed when the cursor enters a consoleField. If the value is *yes*, the line number is specified in the Window field *commentLine*.

**Type:** *Boolean*

**Example:** *hasCommentLine = yes*

**Default:** *no*

**Updatable at run time?** *Yes, but the update has a visual effect only if you open the window after the field is updated*

**highlight**

Specifies the special effects (if any) that are used when displaying the following kinds of output in the window:

- Labels in consoleForms
- Input fields in prompts
- Window border
- Output of system functions such as **ConsoleLib.displayAtPosition**

**Type:** *HighlightKind[]*

**Example:** *highlight = [reverse, underline]*

**Default:** *[noHighLight]*

**Updatable at run time?** *Yes, but the update has a visual effect only if the window is displayed after the field is updated*

Values are as follows:

**noHighlight (the default)**

Causes no special effect. Use of this value overrides any other.

**blink**

Has no effect at this time.

**reverse**

Reverses the text and background colors so that (for example) if the

display has a black background with white letters, the background becomes white and the text becomes black.

#### **underline**

Places an underline under the affected areas. The color of the underline is the color of the text, even if the color of the text has been reversed because you also specified the value **Reverse**.

#### **intensity**

Specifies the strength of the displayed font that is used when displaying the following kinds of output in the window:

- Labels in consoleForms
- Input fields in prompts
- Window border
- Output of system functions such as **ConsoleLib.displayAtPosition**

**Type:** *IntensityKind[]*

**Example:** *intensity = [bold]*

**Default:** *[normalIntensity]*

**Updatable at run time?** *Yes, but the update has a visual effect only if you open the window after the field is updated*

Values are as follows:

#### **normalIntensity (the default)**

Causes no special effect. Use of this value overrides any other.

#### **bold**

Causes the text to appear in a bold-weight font.

#### **dim**

Has no effect at this time. In future, may cause the text to appear with a lessened intensity, as appropriate when all input fields are disabled.

#### **invisible**

Removes any indication that the field is on the form.

#### **menuLine**

Sets the number of the line at which a menu (if any) is displayed in the Window. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on).

**Type:** *INT*

**Example:** *menuLine = 2*

**Default:** *1*

**Updatable at run time?** *Yes, but the update has a visual effect only if you open the window after the field is updated*

The validity of the value is determined only at run time.

#### **messageLine**

Sets the number of the line at which a message (if any) is displayed in the Window. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on).

**Type:** *INT*

**Example:** *messageLine = 3*

**Default:** 2

**Updatable at run time?** *Yes, but the update has a visual effect only if you open the window after the field is updated*

The validity of the value is determined only at run time.

#### **name**

Window name, as used in a programming context in which the name is resolved at run time. It is recommended that the value of the name field be the same as the name of the variable.

**Type:** *String*

**Example:** *name = "myWindow"*

**Default:** *none*

**Updatable at run time?** *No*

#### **position**

The location of the top left corner of the window within the content area of the screen window. The field contains an array of two integers: the line number followed by the column number. The line number is calculated from the top of the screen window's content area (in which case the first line is 1) or (if the value is negative) from the bottom of that area (in which case the last line is -1, the second-to-last is -2, and so on). The column number is calculated from the left of the console window's content area, and the first column is 1.

**Type:** *INT[2]*

**Example:** *position = [2, 3]*

**Default:** *[1,1]*

**Updatable at run time?** *No*

#### **promptLine**

Sets the number of the line at which a prompt (if any) is displayed in the Window. The line number is calculated from the top of the console window's content area or (if the value is negative) from the bottom of that area.

**Type:** *INT*

**Example:** *promptLine = 4*

**Default:** 1

**Updatable at run time?** *Yes, but the update has a visual effect only if you open the window after the field is updated*

The validity of the value is determined only at run time.

#### **size**

An array of two positive integers that represent window dimensions: the number of lines followed by the number of columns.

**Type:** *INT[2]*

**Example:** *size = [24, 80]*

**Default:** *none*

**Updatable at run time?** *No*

A value is required for practical purposes. If you display a window that lacks a value for **size**, the run time presents a window that is too small for content.

If either dimension exceeds the size available in the content area of the screen window, an error occurs at run time.

**Related concepts**

“Console user interface” on page 207

**Related reference**

“ConsoleUI parts and related variables” on page 209

“openUI” on page 726

**Related task**

“Creating an interface with ConsoleUI” on page 208

---

## containerContextDependent

The function part property **containerContextDependent** allows you to extend the name space that is used to resolve function references from within the function part that includes the property. Valid values are *no* (the default) and *yes*.

It is recommended that you avoid using this capability when you develop new code. The property is primarily available for migrating programs from VisualAge Generator. If you set this property to *yes*, however, the implications are as follows:

- If the usual steps of a name search do not resolve a reference at editing time, the EGL editor does not flag the unresolved references as errors.
- If the usual steps of a name search do not resolve a reference at generation time, the search continues by reviewing the name space of the program, library, or PageHandler that contains the function part.
- If you have declared a function at the top level of an EGL source file rather than physically inside a container (a program, PageHandler, or library), that function can invoke library functions only if the following situation is in effect:
  - The container includes a use statement that refers to the library
  - In the invoking function, the property **containerContextDependent** is set to *yes*

**Related concepts**

“References to parts” on page 23

**Related reference**

“Function part in EGL source format” on page 621

]“Use declaration” on page 1091

---

## Database authorization and table names

An authorization ID is a character string that is passed to the database manager when a connection is established between the database manager and a program, whether the program prepares another program or allows end-user access to SQL tables. The character string is the user identifier that is required to check the database-access authorization held by the preparer or end user.

The source of the authorization ID depends on the system where database access occurs.

The situation for EGL-generated Java programs is as follows:

- The authorization ID is one obtained by the database manager when a connection was established between the database manager and the program:



- In relation to the default database, the authorization ID is the value specified for the Java runtime property **vgj.jdbc.default.userid**
- When you invoke the system function **sysLib.connect** or **VGLib.connectionService**, the authorization ID is the value specified for the **userID** parameter

The authorization ID may be used when you specify a table name. In that case, you can specify a table-name qualifier, in accordance with this syntax:

*tableOwner.myTable*

*tableOwner*

A qualifier that is known to the database manager and that is necessary to identify the table. The qualifier at table creation is the authorization ID of the person who created the table.

*myTable*

The table name.

For more information on authorization IDs, consult your database manager documentation.

### Related concepts

“Dynamic SQL” on page 288  
 “Java runtime properties” on page 431  
 “SQL support” on page 277

### Related reference

“Java runtime properties (details)” on page 642  
 “SQL record part in EGL source format” on page 877  
 “connect()” on page 1025  
 “connectionService()” on page 1047

---

## Data conversion

Because of differences in how data is interpreted in different runtime environments, your program may need to convert the data that passes from one environment to another. Data conversion occurs at Java run time.

Your code also uses a conversion table in the following runtime situations:

- Your EGL-generated Java code calls a program on CICS for z/OS.  
 In this case, you can specify the conversion table in a `callLink` element that refers to the called program. Alternatively, you can indicate (in that `callLink` element) that the system variable `sysVar.callConversionTable` identifies the conversion table at run time.
- An EGL-generated program (on a platform that supports the EBCDIC character set) transfers asynchronously to a program on a platform that supports the ASCII character set, as might occur when the transferring program invokes the system function `sysLib.startTransaction`.

In this case, you can specify the conversion table in a `asynchLink` element that refers to the program to which control is transferred. Alternatively, you can indicate (in that `asynchLink` element) that the system variable `sysVar.callConversionTable` identifies the conversion table at run time.

- An EGL-generated Java program shows a text or print form that includes series of Arabic or Hebrew characters; or presents a text form that accepts a series of such characters from the user.

In these cases, you specify the bidirectional conversion table in the system variable `sysVar.formConversionTable`.

You would use runtime conversion, for example, if your code places values into one of two redefined records, each of which refers to the same area of memory as a record that is passed to another program. Assume that the characteristics of the data that you pass would be different, depending on the redefined record to which you assign values. In this case, the requirements of data conversion cannot be known at generation time.

The next sections provide the following details:

- 
- “Data conversion when the invoker is Java code”
- “Conversion algorithm” on page 560

## Data conversion when the invoker is Java code

The following rules pertain to Java code:

- When a generated Java program or wrapper invokes a generated Java program, conversion occurs in the caller, in accordance with a set of EGL classes invoked at runtime. It is sufficient to request no conversion at all in most cases, even if the caller is accessing a remote platform that uses a code page that is different from the code page used by the invoker. You must specify a conversion table, however, in the following situation:
  - The caller is Java code and is on a machine that supports one code page
  - The called program is non-Java and is on a machine that supports another code page

The table name in this case is a symbol that indicates the kind of conversion that is required at run time.

- When a generated Java program accesses a remote MQSeries message queue, conversion occurs in the invoker, in accordance with a set of EGL classes invoked at run time. If the caller is accessing a remote platform that uses a code page that is different from the code page used by the invoker, specify a conversion table in the association element that refers to the MQSeries message queue.

The next table lists the conversion tables that can be accessed by generated Java code at run time. Each name has the format CSOJx:

- x Represents the code page number on the invoked platform. Each number is specified in the *Character Data Representation Architecture Reference and Registry*, SC09-2190. The registry identifies the coded character sets supported by the conversion tables.

Language	Platform of Invoked Program		
	UNIX	Windows 2000/NT/XP	z/OS UNIX System Services or iSeries Java
Arabic	CSOJ1046	CSOJ1256	CSOJ420
Chinese, simplified	CSOJ1381	CSOJ1386	CSOJ1388

Language	Platform of Invoked Program		
	UNIX	Windows 2000/NT/XP	z/OS UNIX System Services or iSeries Java
Chinese, traditional	CSOJ950	CSOJ950	CSOJ1371
Cyrillic	CSOJ866	CSOJ1251	CSOJ1025
Danish	CSOJ850	CSOJ850	CSOJ277
Eastern European	CSOJ852	CSOJ1250	CSOJ870
English (UK)	CSOJ850	CSOJ1252	CSOJ285
English (US)	CSOJ850	CSOJ1252	CSOJ037
French	CSOJ850	CSOJ1252	CSOJ297
German	CSOJ850	CSOJ1252	CSOJ273
Greek	CSOJ813	CSOJ1253	CSOJ875
Hebrew	CSOJ856	CSOJ1255	CSOJ424
Japanese	CSOJ943	CSOJ943	CSOJ1390 (Katakana SBCS), CSOJ1399 (Latin SBCS)
Korean	CSOJ949	CSOJ949	CSOJ1364
Portuguese	CSOJ850	CSOJ1252	CSOJ037
Spanish	CSOJ850	CSOJ1252	CSOJ284
Swedish	CSOJ850	CSOJ1252	CSOJ278
Swiss German	CSOJ850	CSOJ1252	CSOJ500
Turkish	CSOJ920	CSOJ1254	CSOJ1026

If you do not specify a value for the conversion table in the linkage options part when you are calling a program from Java, the default conversion tables are those for English (US).

## Conversion algorithm

Data conversion of records and structures is based on the declarations of the structure items that lack a substructure.

Data of type CHAR, DBCHAR, or MBCHAR is converted in accordance with the Java conversion tables (for conversion that occurs in an EGL-generated invoker).

No conversion is performed for filler data items (data items that have no name) or for data items of type DECIMAL, PACF, HEX, or UNICODE.

On EBCDIC-to-ASCII conversion for MBCHAR data, the conversion routine deletes shift-out/shift-in (SO/SI) characters and inserts an equivalent number of blanks at the end of the data item. On ASCII-to-EBCDIC conversion, the conversion routine inserts SO/SI characters around double-byte strings and truncates the value at the last valid character that can fit in the field. If the MBCHAR field is in a variable length record and the current record end is in the MBCHAR field, the record length is adjusted to reflect the insertion or deletion of SO/SI characters. The record length indicates where the current record ends.

For data items of type BIN, the conversion routine reverses the byte order of the item if the caller or called platform uses Intel binary format and the other platform does not.

For data items of type NUM or NUMC items, the conversion routine converts all but the last byte using the CHAR algorithm. The sign half-byte (the first half byte of the last byte in the field) is converted according to the hexadecimal values shown in the next table.

EBCDIC for type NUM	EBCDIC for type NUMC	ASCII
F (positive sign)	C	3
D (negative sign)	D	7

#### Related reference

“Association elements” on page 457

“Bidirectional language text”

“callLink element” on page 499

“convert()” on page 1027

“callConversionTable” on page 1066

## Bidirectional language text

Bidirectional (bidi) languages such as Arabic and Hebrew are languages in which the text is presented to the user ordered from right to left, but numbers and Latin alphabetic strings within the text are presented left to right. In addition, the order in which characters appear within program variables can vary. The text is usually stored in *logical* order, the order in which the characters are entered in the input field.

These differences in ordering and in other associated presentation characteristics require the program to have the ability to convert bi-directional text strings from one format to another. The bidi conversion attributes are specified in a bidi conversion table (.bct) file created separately from the program. The program references the name of the conversion table to indicate how attribute conversion should be performed.

In all cases, the bidi conversion table reference is specified as the 1 to 8 character file name without the .bct extension. For example, if you have created a bidi conversion table named mybct.bct, you can set the value of formConversionTable in a program by adding the following statement at the beginning of the program:

```
sysVar.formConversionTable = "mybct.bct" ;
```

Your tasks are as follows:

- Create bidi conversion tables that specify the transformations that should occur. Note that different tables are needed for converting data to be displayed in a text or print form .
- When generating a program that uses text or print forms with bidi language text, add a statement to the program that assigns the conversion table name to the system function sysVar.formConversionTable before showing the form.

You build the bidi conversion table file using the bidi conversion table wizard plugin, which is in the file BidiConversionTable.zip:

1. Download the file BidiConversionTable.zip.
2. Unzip the file into your workbench directory
3. To begin running the wizard, click **File > New > Other > BidiConversionTable**.

The name of a table used with EGL programs must have eight characters or less and must have the .bct extension.

4. While running the wizard, press F1 for help in choosing the correct options for creating the table.

#### Related reference

“Data conversion” on page 558

“callConversionTable” on page 1066

---

## Data conversions between WSDL and EGL

This topic shows the default mapping of WSDL definitions to and from EGL types. You can override the default value for a given EGL field by using the primitive field-level property **@xsd**.

The next table shows the default mapping of built-in XML subschema definition (XSD) simple types to the EGL types.

XSD simple type	EGL type
xsd:string	STRING
xsd:integer	NUM
xsd:int	INT
xsd:long	BIGINT
xsd:short	SMALLINT
xsd:decimal	DECIMAL
xsd:float	SMALLFLOAT
xsd:double	FLOAT
xsd:boolean	SMALLINT
xsd:byte	HEX
xsd:unsignedInt	BIGINT
xsd:unsignedShort	INT
xsd:unsignedByte	HEX
xsd:QName	STRING
xsd:dateTime	TIMESTAMP
xsd:date	DATE
xsd:time	TIME
xsd:anyURI	STRING
xsd:base64Binary	HEX[]
xsd:hexBinary	HEX[]

XSD simple type	EGL type
xsd:anySimpleType	STRING
xsd:duration	INTERVAL
xsd:gYearMonth	STRING
xsd:gYear	STRING
xsd:gMonthDay	STRING
xsd:gDay	STRING
xsd:gMonth	STRING
xsd:normalizedString	STRING
xsd:token	STRING
xsd:language	STRING
xsd:Name	STRING
xsd:NCName	STRING
xsd:ID	STRING
xsd:NMTOKEN	STRING
xsd:NMTOKENS	STRING[]
xsd:nonPositiveInteger	DECIMAL
xsd:negativeInteger	DECIMAL
xsd:nonNegativeInteger	DECIMAL
xsd:unsignedLong	DECIMAL
xsd:positiveInteger	DECIMAL

The next table shows the default mapping of SOAP encoded types to the EGL types.

SOAP encoded type	EGL type
soapenc:string	STRING
soapenc:boolean	SMALLINT
soapenc:float	SMALLFLOAT
soapenc:double	FLOAT
soapenc:decimal	DECIMAL
soapenc:int	INT
soapenc:short	SMALLINT
soapenc:byte	HEX
soapenc:base64	HEX[]

The next table shows the default mapping of EGL types to XSD types. A derived XSD simple type is created for each entry that says *derived from*.

EGL type	XML schema mapping
BIGINT	xsd:long
BIN(4) without decimals	xsd:short
BIN(9) without decimals	xsd:int

EGL type	XML schema mapping
BIN(18) without decimals	xsd:long
BIN with decimals	derived from xsd:decimal
CHAR	derived from xsd:string
DATE	xsd:date
DBCHAR	derived from xsd:string
DECIMAL	derived from xsd:decimal
FLOAT	xsd:double
HEX	derived type xsd:hexBinary
INT	xsd:int
INTERVAL	xsd:duration
MBCHAR	derived from xsd:string
MONEY	derived from xsd:decimal
NUM	derived from xsd:decimal
NUMC	derived from xsd:decimal
PACF	derived from xsd:decimal
SMALLFLOAT	xsd:float
SMALLINT	xsd:short
STRING	xsd:string
TIME	xsd:time
TIMESTAMP	xsd:dateTime
UNICODE	derived from xsd:string

### Related concepts

“EGL interfaces” on page 151

“EGL services and Web services” on page 158

### Related tasks

“Creating an EGL Interface part” on page 150

“Creating an Interface part from a Service part” on page 154

“Creating an EGL Service part” on page 157

### Related reference

“@xsd” on page 798

---

## Data initialization

EGL handles data initialization as follows:

- In some cases you can code an *initializer*, which is an equal sign followed by a literal--
  - When you define a fixed record part, you can code an initializer for any lowest-level field:

```
Record myRecordPart type basicRecord
  10 myField CHAR(5);
  20 myField01 CHAR(1) = "1";
  20 myField02 CHAR(1) = "2";
  20 myArray01 CHAR(1)[3] = ["a", "b", "c"];
```

```

        // the following entry assigns "z" to the first element
        // and (in Java code) blanks to the rest
        20 myArray02 CHAR(1)[3] = ["z"];
    end

```

This rule also applies to form fields; however, you cannot specify an initializer in a DataTable.

- When you declare a primitive field, you can code an initializer as well, whether the field is a parameter, a field in a non-fixed record, or another variable:

```

Record myRecordPart type basicRecord
    myRecField INT = 2;
end

```

```

Program myProgram (myField03 INT = 3)
    myField04 STRING = "EGL";

```

```

    function main()

        // myRecord.myRecField = 2
        myRecord myRecordPart;
    end
end

```

- An initializer in a record that redefines another record has no effect at declaration time, but is used if your code invokes a **set** statement of the form *set record initial*:

```

Record partA
    10 aa char(4) = "abcd";
end

```

```

Record partB
    10 bb char(4) = "1234";
end

```

```

Program Example
    A partA;
    B partB { redefines="A" };

```

```

    function main()

        // each of the next statements writes "abcd"
        writeStdOut( A.aa );
        writeStdOut( B.bb );

        // sets the memory area in a way that reflects the definition of record partB
        set B initial;

        // each of the next statements writes "1234"
        writeStdOut( A.aa );
        writeStdOut( B.bb );
    end
end

```

- EGL also initializes memory with the preset values that are described later, in the following cases--
  - Your logic invokes some variations of the **set** statement, as described in *set*.
  - The **initialized** property of a primitive field is set to *yes*, as is possible only if the field is outside of a non-fixed record.
  - A local or global record is in a Java program, PageHandler, or record handler.



In a fixed structure, only the lowest-level structure fields are considered. If a structure field of type HEX is subordinate to a structure item of type CHAR, for example, the memory area is initialized with binary zeros, as is appropriate for HEX initialization.

Records or fields that are received as program or function parameters are never initialized automatically.

The next table gives details on the initialization values.

Primitive type	Initialization value
ANY	Variable is of an undefined type
BIN (and the integer types), HEX, FLOAT, SMALLFLOAT	Binary zeros
CHAR, MBCHAR	Single-byte blanks
DATE, TIME, TIMESTAMP	Current value of the machine clock (for the number of bytes required by the mask, in the case of TIMESTAMP)
DBCHAR	Double-byte blanks
DECIMAL, MONEY, NUM, NUMC, PACF	Numeric zeros
INTERVAL	Numeric zeros (for the number of bytes required by the mask), preceded by a plus sign
UNICODE	Unicode blanks (each of which is hexadecimal 0020)

#### Related concepts

“Function part” on page 150

“DataItem part” on page 133

“Program part” on page 148

“Record parts” on page 135

“Fixed structure” on page 27

#### Related reference

“EGL statements” on page 88

“Primitive types” on page 34

“set” on page 742

---

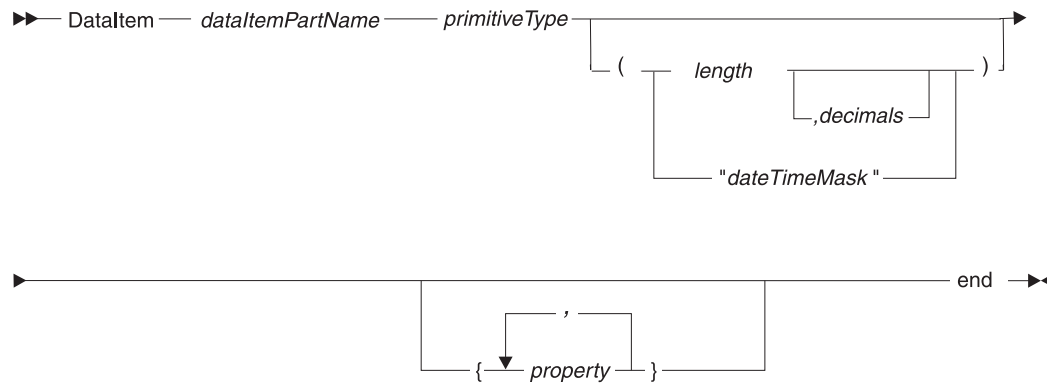
## DataItem part in EGL source format

You declare a DataItem part in an EGL source file, which is described in *EGL source format*.

An example of a DataItem part is as follows:

```
DataItem myDataItemPart
  BIN(9,2)
end
```

The syntax diagram for a dataItem part is as follows:



### **DataItem dataItemPartName ... end**

Identifies the part as a dataItem part and specifies the name. For rules, see *naming conventions*.

#### *primitiveType*

The primitive type assigned to the dataItem part.

#### *length*

An integer that reflects the length of the dataItem part. The value of any variable that is based on the part includes the specified number of characters or digits.

#### *decimals*

For a numeric, fixed type other than MONEY (specifically, BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

#### *"dateTimeMask"*

For items of type INTERVAL or TIMESTAMP, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the item value. The mask is present with the data at run time.

#### *property*

An item property, as described in *Overview of EGL properties*.

### **Related concepts**

- "DataItem part" on page 133
- "EGL projects, packages, and files" on page 15
- "Overview of EGL properties" on page 64
- "References to parts" on page 23
- "Parts" on page 19

### **Related tasks**

- "Syntax diagram for EGL statements and commands" on page 884

### **Related reference**

- "EGL source format" on page 586
- "Function part in EGL source format" on page 621
- "Indexed record part in EGL source format" on page 632
- "MQ record part in EGL source format" on page 769
- "Naming conventions" on page 778
- "Primitive types" on page 34
- "Program part in EGL source format" on page 841

“Relative record part in EGL source format” on page 865

“Serial record part in EGL source format” on page 868

“SQL record part in EGL source format” on page 877

---

## DataTable part in EGL source format

You declare a dataTable part in an EGL source file, which is described in *EGL projects, packages, and files*. This part is a generatable part, which means that it must be at the top level of the file and must have the same name as the file.

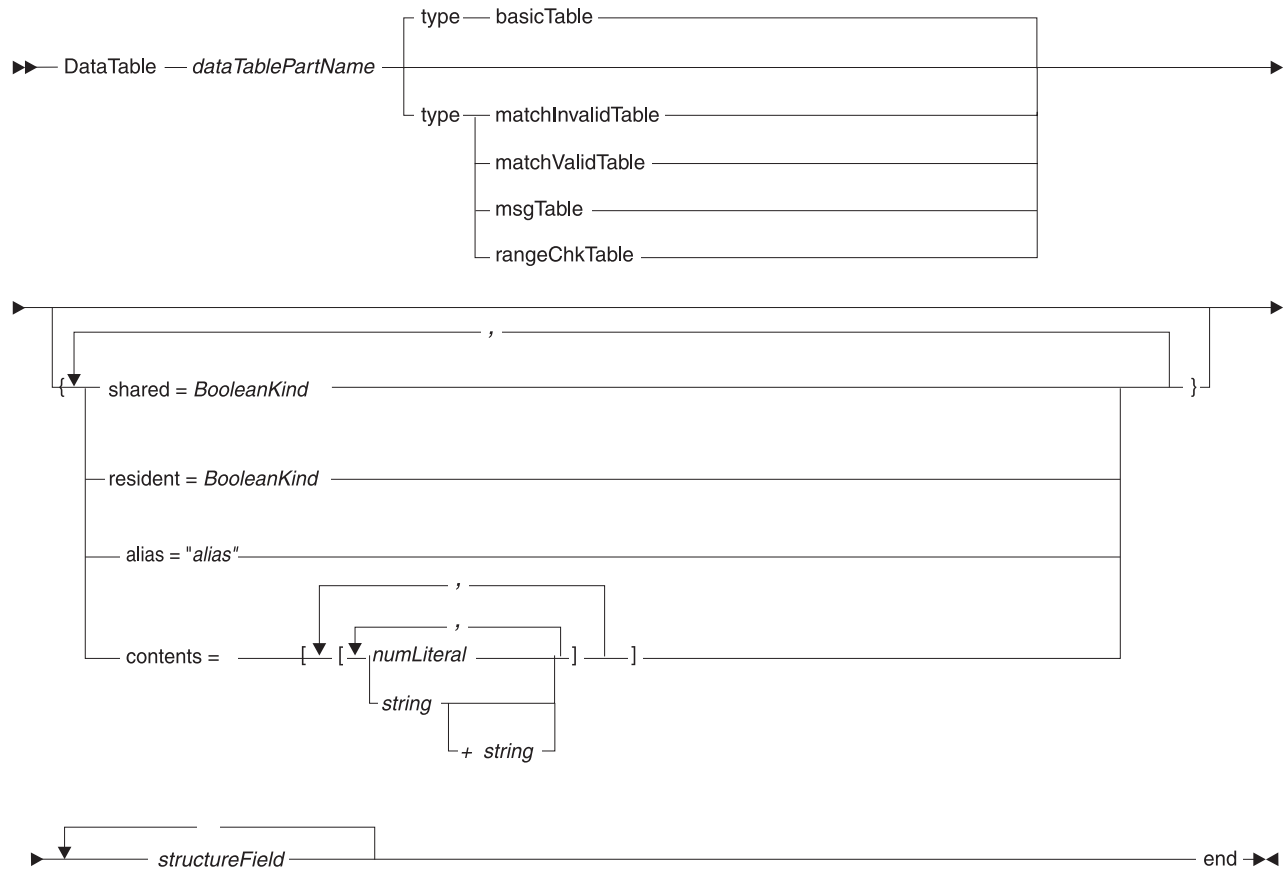
A dataTable is related to a program by the program’s use declaration or (in the case of the program’s only message table) by the program’s **msgTablePrefix** property. A dataTable is related to a pageHandler by the pageHandler’s use declaration.

An example of a dataTable part is as follows:

```
DataTable myDataTablePart type basicTable
{
  { shared = yes }
  myColumn1 char(10);
  myColumn2 char(10);
  myColumn3 char(10);

  { contents = [
    [ "row1 col1", "row1 col2", "row1 " + "col3" ] ,
    [ "row2 col1", "row2 col2", "row2 " + "col3" ] ,
    [ "row3 col1", "row3 col2", "row3 col3"      ]
  ]
}
end
```

The syntax diagram for a dataTable part is as follows:



### **DataTable** *dataTablePartName* ... **end**

Identifies the part as a dataTable and specifies the part name. For the rules of naming, see *Naming conventions*.

### **basicTable** (the default)

Contains information that is used in the program logic; for example, a list of countries and related codes.

### **matchInvalidTable**

Is specified in the **validatorDataTable** property of a text field to indicate that the user's input must be different from any value in the first column of the dataTable. The EGL runtime acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorDataTable** property
- Retrieves the message identified by the **validatorDataTableMsgKey** property that is specific to the text field
- Displays the message in the text field identified in the form-specific **msgField** property

### **matchValidTable**

Is specified in the **validatorDataTable** property of a text field to indicate that the user's input must match a value in the first column of the dataTable. The EGL runtime acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorDataTable** property
- Retrieves the message identified by the **validatorDataTableMsgKey** property that is specific to the text field
- Displays the message in the field identified in the form-specific **msgField** property

### msgTable

Contains runtime messages. A message is presented in the following circumstance:

- The table is the message table for the program. The association of DataTable to program occurs if the program property **msgTablePrefix** references the *table prefix*, which is the first one to four characters in the name of the DataTable. When you name the message table, you include a three-character code to represent the national language, as shown in the table displayed next.

Language	National language code
Brazilian Portugese	PTB
Chinese, simplified	CHS
Chinese, traditional	CHT
English, uppercasse	ENP
English, USA	ENU
French	FRA
German	DEU
Italian	ITA
Japanese, Katakana (single-byte character set)	JPN
Korean	KOR
Spanish	ESP
Swiss German	DES

- The program retrieves and presents a message by one of two mechanisms, as described in *ConverseLib.displayMsgNum* and *ConverseLib.validationFailed*.

### rangeChkTable

Is specified in the **validatorDataTable** property of a text field to indicate that the user's input must match a value that is between the values in the first and second column of at least one data-table row. (The range is inclusive; the user's input is valid if it matches a value in the first or second column of any row.)

The EGL runtime acts as follows in response to a validation failure:

- Accesses the table referenced in the **validatorDataTable** property
- Retrieves the message identified by the **validatorDataTableMsgKey** property that is specific to the text field
- Displays the message in the field identified in the form-specific **msgField** property

### "alias"

A string that is incorporated into the names of generated output. If you do not specify an alias, the dataTable name is used instead.

### shared

Indicates whether the same instance of a dataTable is used by multiple programs in the same run unit. Valid values are *yes* and *no* (the default). If the value of **shared** is *no*, each program in the run unit has a unique copy of the dataTable.

The property indicates whether the same instance of a dataTable is used by every program in the same run unit. If the value of **shared** is *no*, each program in the run unit has a unique copy of the dataTable.

Changes made at run time are visible to every program that has access to the dataTable, and the changes remain until the dataTable is unloaded. In most cases, the value of the **resident** property (described later) determines when the dataTable is unloaded; for details, see the description of that property.

### **resident**

Indicates whether the dataTable is kept in memory even after every program that accessed the dataTable has ended.

Valid values are *yes* and *no*. The default is *no*.

If you set the **resident** property to *yes*, the dataTable is shared regardless of the value of **shared**.

The benefits of making a dataTable resident are as follows:

- The dataTable retains any values written to it by programs that ran previously
- The table is available for immediate access without additional load processing

A resident dataTable remains loaded until the run unit ends. A non-resident dataTable, however, is unloaded when the program that uses it ends.

**Note:** A dataTable is loaded into memory (if necessary) at a program's first access, and not when the EGL runtime processes a use declaration.

### **contents**

The value of the dataTable cells, each of which is one of the following kinds:

- A numeric literal
- A string literal or a concatenation of string literals

The kind of content in a given row must be compatible with the top-level structure fields, each of which represents a column definition.

### *structureField*

A structure field, as described in *Structure field in EGL source format*.

### **Related concepts**

"DataTable" on page 176

"EGL projects, packages, and files" on page 15

"Run unit" on page 866

### **Related reference**

"Naming conventions" on page 778

"Structure field in EGL source format" on page 880

"displayMsgNum()" on page 917

"validationFailed()" on page 918

"Use declaration" on page 1091

---

## **EGL build path and eglpath**

Each EGL project and EGL Web project is associated with an EGL build path so that the project can reference parts in other projects. For details on when the EGL build path is used and on why the order of build-path entries is important, see *References to parts*.

When you specify the EGL build path, you can choose to *export* one or more of the projects that are listed in the build path. Then, when a project refers to the project being declared, each of the exported projects is made available to the referencing project, as in the following example:

- The EGL build path for project A comprises the following projects, in order:

A, B, C, D

Projects B and D are exported.

- The EGL build path for project L comprises the following projects, in order:

L, J, A, Z

- The effective build path for project L also includes the projects that were exported from project A. In this case, the EGL build path for project L is effectively as follows:

L, J, A, B, D, Z

The exported projects are placed after the project that exports them, in the order in which the projects are listed in the build path of the exporting project.

The build path of a project always includes the project itself, which is usually first in build-path order, as is recommended. If you have multiple EGL source folders in your project, all must be listed in the EGL build path for that project, and the order of those folders is used by any project that refers to your project.

*It is strongly recommended that you avoid having identically named packages in different projects or in different folders of the same project.*

If you generate in the EGL SDK, the situation is as follows:

- Project information is not available.
- The command-line argument *eglpath* replaces the functionality of the EGL build path. *eglpath* is a list of operating-system directories that are searched when the EGL SDK attempts to resolve a part reference.
- The rules for when *eglpath* is used are equivalent to the rules for when the EGL build path is used; however, you cannot export directories as you can export projects.

*When you use the EGL SDK, it is strongly recommended that you avoid having identically named packages in different directories.*

#### **Related concepts**

“Generation from the EGL Software Development Kit (SDK)” on page 418

“References to parts” on page 23

#### **Related tasks**

“Generating from the EGL Software Development Kit (SDK)” on page 418

#### **Related reference**

“EGLSDK” on page 583

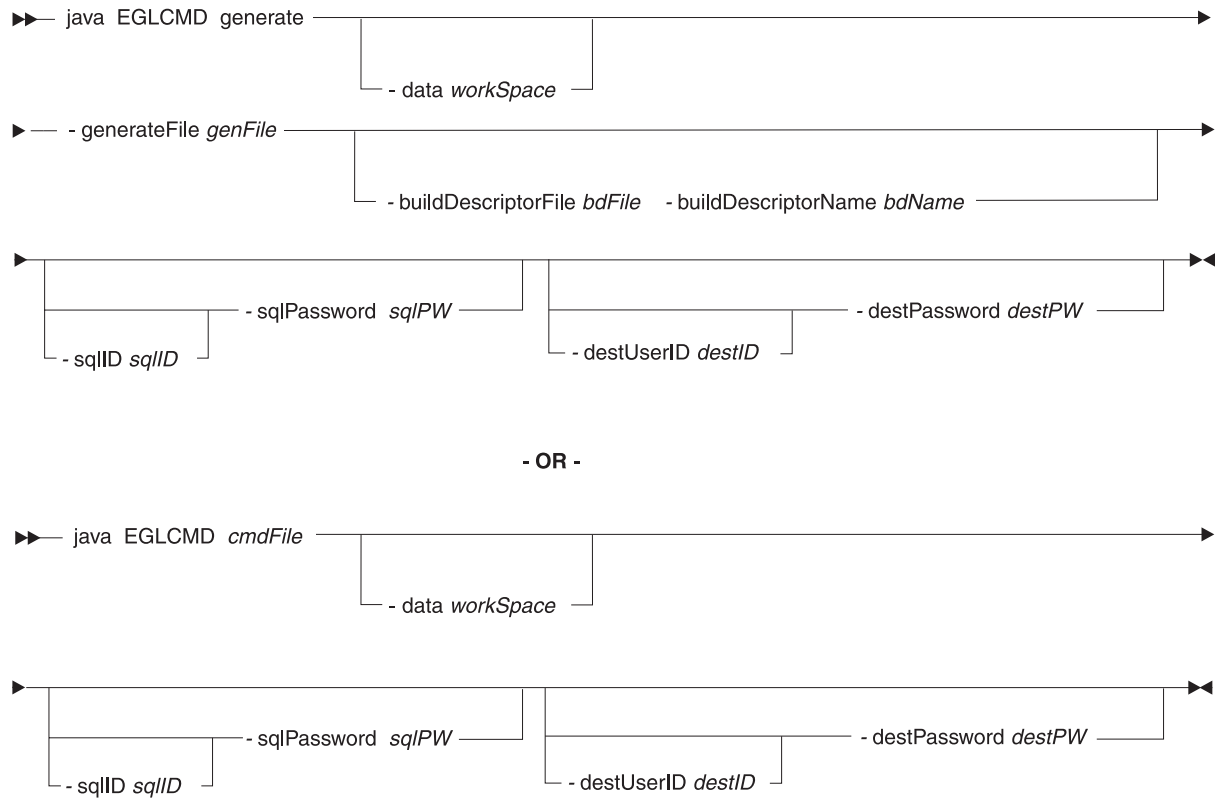
---

## **EGLCMD**

The command EGLCMD gives you access to the Workbench batch interface, as described in *Generation from the workbench batch interface*.

## Syntax

The syntax for invoking EGLCMD is as follows:



### **generate**

Indicates that the command itself references the EGL source file and build descriptor part that are used to generate output. In this case, the command EGLCMD does not reference a command file.

#### **-data** *workspace*

Specifies the absolute or relative path of the workspace directory. Relative paths are relative to the directory in which you run the command.

If you do not specify a value, the command accesses the Eclipse default workspace.

Embed the path in double quotes.

#### **cmdFile**

Specifies the absolute or relative path of the file described in *EGL command file*. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

The command file must be in your workspace; otherwise, use the Eclipse import process to import the file and then rerun EGLCMD.

#### **-generateFile** *genFile*

Specifies the absolute or relative path of the EGL file that contains the part you want to process. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.



**-buildDescriptorFile** *bdFile*

Specifies the absolute or relative path of the build file that contains the build descriptor. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

**-buildDescriptorName** *bdName*

Specifies the name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file.

**-sqlID** *sqlID*

Sets the value of build descriptor option sqlID.

**-sqlPassword** *sqlPW*

Sets the value of build descriptor option sqlPassword.

**-destUserid** *destID*

Sets the value of build descriptor option destUserID.

**-destPassword** *destPW*

Sets the value of build descriptor option destPassword.

Build descriptor options that you specify when invoking the command EGLCMD take precedence over options in the build descriptor (if any) that is listed in the EGL command file.

## Examples

In the commands that follow, each multiline example belongs on a single line:

```
java EGLCMD "commandfile.xml"
```

```
java EGLCMD "commandfile.xml" -data "c:\myWorkSpace"
```

```
java EGLCMD generate
  -generateFile "c:\myProg.eglpgm"
  -data "myWorkSpace"
  -buildDescriptorFile "c:\myBuild.eglbld"
  -buildDescriptorName myBuildDescriptor
```

```
java EGLCMD "myCommand.xml"
  -data "my WorkSpace"
  -sqlID myID -sqlPassword myPW
  -destUserID myUserID -destPassword myPass
```

### Related concepts

"Generation from the workbench batch interface" on page 417

### Related tasks

"Generating from the workbench batch interface" on page 417

"Syntax diagram for EGL statements and commands" on page 884

### Related reference

"destPassword" on page 475

"destUserID" on page 475

"sqlID" on page 491

"sqlPassword" on page 492

---

## EGL command file

An EGL command file indicates what EGL files you wish to process when you generate output outside of the workbench, whether you are using the workbench batch interface (command EGLCMD) or the EGL SDK (command EGLSDK). You can create the file in either of two ways:

- By hand, according to the rules described later; or
- By using the EGL Generation wizard, as described in *Generating in the workbench*.

The command file is an XML file, and the file name must have the extension .xml, in any combination of uppercase and lowercase letters. The file content must conform to the following document type definition (DTD):

```
installationDir\egl\eclipse\plugins\  
com.ibm.etools.egl.utilities_version\  
dtd\eglcommands_5_1.dtd
```

*installationDir*

The product installation directory, such as C:\Program Files\IBM\RSPD\6.0. If you installed and kept a Rational Developer product before installing the product that you are using now, you may need to specify the directory that was used in the earlier install.

*version*

The installed version of the plugin; for example, 6.0.0

The following table shows the elements and attributes supported by the DTD. The element and attribute names are case sensitive.

Element	Attribute	Attribute value
<b>EGLCOMMANDS</b> (required)	<b>eglpath</b>	<p>As described in <i>eglpath</i>, the eglpath attribute identifies directories to search when EGL uses an import statement to resolve the name of a part. The attribute is optional and if present, references a quoted string that has one or more directory names, each separated from the next by a semicolon.</p> <p>The attribute is used only if the command EGLSDK is referencing the command file. If the command EGLCMD is in use, the value of eglpath is ignored; instead, import statements are resolved in accordance with the EGL project path, as described in <i>Import</i>.</p>

Element	Attribute	Attribute value
<b>buildDescriptor</b> (optional; you can avoid specifying this value if you are using a master build descriptor, as described in <i>Build descriptor part</i> )	<b>name</b>	The name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file.  Build descriptor options that you specify when invoking EGLCMD or EGLSDK take precedence over options in the build descriptor (if any) that is listed in the EGL command file.
	<b>file</b>	The absolute or relative path of the EGL file that contains the build descriptor. Relative paths specified for EGLCMD are relative to the path name of the Enterprise Developer workspace. Relative paths specified for EGLSDK are relative to the directory in which you run the command.  The path must be in double quotes if the path includes a space.
<b>generate</b> (optional)	<b>file</b>	The absolute or relative path of the EGL file that contains the part you want to process. Relative paths specified for EGLCMD are relative to the path name of the Enterprise Developer workspace. Relative paths specified for EGLSDK are relative to the directory in which you run the command.  The path must be in double quotes if the path includes a space.  If you omit the file attribute, no generation occurs.

## Examples of command files

This section shows two command files. The results produced by either file are the same whether you use the EGLCMD command or the EGLSDK command, if you run the EGLSDK command in the directory where the EGL source files reside.

The following command file contains a generate command that uses the build descriptor myBDescPart to generate the program myProgram.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGLCOMMANDS PUBLIC "-//IBM//DTD EGLCOMMANDS 5.1//EN" "">
<EGLCOMMANDS eglpath="C:\mydata\entdev\workspace\projectinteract">
  <generate file="projectinteract\myProgram.eglpgm">
    <buildDescriptor name="myBDescPart" file="projectinteract\mybdesc.eglbld"/>
  </generate>
</EGLCOMMANDS>
```

The next example contains two generate commands, both of which implicitly use a master build descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGLCOMMANDS PUBLIC "-//IBM//DTD EGLCOMMANDS 5.1//EN" "">
<EGLCOMMANDS eglpath="C:\mydata\entdev\workspace\projecttrade">
  <generate file="projecttrade\program2.eglpgm"/>
  <generate file="projecttrade\program3.eglpgm"/>
</EGLCOMMANDS>
```

### Related concepts

“Build descriptor part” on page 383

“Generation from the EGL Software Development Kit (SDK)” on page 418  
“Generation from the workbench batch interface” on page 417  
“Import” on page 33

#### Related tasks

“Generating from the EGL Software Development Kit (SDK)” on page 418  
“Generating from the workbench batch interface” on page 417  
“Generating in the workbench” on page 414

#### Related reference

“EGLCMD” on page 572  
“EGL command file” on page 575  
“EGL build path and eglpath” on page 571  
“EGLSDK” on page 583

---

## EGL editor

To change an EGL source file (extension .egl), work in the EGL source editor, which guides you with content assist.

To change an EGL build file (extension .eglbld), follow one of these procedures:

- Creating a build file
- 
- Adding a build descriptor part
- Adding a linkage options part
- 
- Adding a resource associations part

#### Related concepts

“EGL projects, packages, and files” on page 15  
“Parts” on page 19

#### Related reference

“Content assist in EGL”

## Content assist in EGL

The EGL editor provides *content assist*, which proposes information that you can add to your source file. With a keystroke or two, you can complete the name of a part, variable, or function or can place a *template* (the outline of a part) into your source file.

The keystroke that activates content assist is **Ctrl + Space**.

#### Related concepts

“Source assist in EGL”

#### Related tasks

“Setting preferences for templates” on page 119  
“Using the EGL templates with content assist” on page 131

## Source assist in EGL

The EGL editor provides a *source assistant* that helps you edit dataItem parts. The source assistant displays all of the valid properties for a dataItem and lets you

enter values for those properties. The source assistant validates the values you enter. When you are finished, the source assistant writes the EGL code to define the `dataItem` part.

See *Editing a dataItem part with the source assistant*.

#### Related concepts

"DataItem part" on page 133

#### Related tasks

"Editing a dataItem part with the source assistant" on page 134

#### Related reference

"Content assist in EGL" on page 577

---

## Enumerations in EGL

In some cases in EGL, the values of a property or field are restricted to the values of a particular *enumeration*, which is a category of predefined values. The property **color**, for example, accepts a value of the enumeration **ColorKind**, and valid values of that enumeration include *white* and *red*.

You can qualify an enumeration value with the enumeration name, so the preceding values can be stated as *ColorKind.white* and *ColorKind.red*. However, you need to qualify the enumeration value only when your code has access to a variable or constant whose name is the same as the enumeration value. If a variable named *red* is in scope, for example, the symbol *red* refers to the variable rather than to the enumeration value.

The following list of enumerations includes the enumeration values; but explanations of those values occur elsewhere, in the context of the property or field in which the enumeration is meaningful:

#### **AlignKind**

- center
- left
- none
- right

#### **Boolean**

- yes
- no

#### **CallingConventionKind**

- I4GL
- Library

#### **CaseFormatKind**

- defaultCase
- lower
- upper

#### **ColorKind**

- black (as is valid only for console fields)
- blue

cyan  
defaultColor  
green  
magenta  
red  
yellow  
white

**CommTypeKind**

LOCAL  
TCPIP

**DataSource**

databaseConnection  
reportData  
sqlStatement

**DeviceTypeKind**

doubleByte  
singleByte

**DisplayUseKind**

button  
hyperlink  
input  
output  
secret  
table

**EventKind**

AFTER\_DELETE  
AFTER\_FIELD  
AFTER\_OPENUI  
AFTER\_INSERT  
AFTER\_ROW  
BEFORE\_DELETE  
BEFORE\_FIELD  
BEFORE\_OPENUI  
BEFORE\_INSERT  
BEFORE\_ROW  
ON\_KEY  
MENU\_ACTION

**ExportFormat**

html  
pdf  
text  
xml

**HighlightKind**

- blink
- defaultHighlight
- noHighlight
- reverse
- underline

**IndexOrientationKind**

- across
- down

**IntensityKind**

- bold
- defaultHighlight
- dim
- invisible
- normalIntensity

**LineWrapKind**

- character
- compress (as is valid only for console fields)
- word

**OutlineKind**

- bottom
- left
- right
- top

**Note:** `sysLib.box` is a constant that equates to `[left,right,top,bottom]`.  
`sysLib.noOutline` is a constant that means there is no outlining.

**PCBKind**

- DB
- GSAM
- TP

**PfKeyKind**

- `pfn`, where  $(1 \leq n \leq 24)$

**ProtectKind**

- skip
- no
- yes

**SelectTypeKind**

- index
- value

**SignKind**

- leading
- none
- parens
- trailing

### **SignKind**

leading  
none  
parens  
trailing

### **UITypeKind**

hidden  
input  
inputOutput  
none  
output  
programLink  
uiForm

### **WindowAttributeKind**

color  
commentLine  
errorLine  
formLine  
highlight  
intensity  
menuLine  
messageLine  
promptLine

### **Related concepts**

“Overview of EGL properties” on page 64

“References to variables in EGL” on page 59

---

## **EGL reserved words**

EGL includes two categories of reserved words:

- Words that are reserved for specific uses except when you are working on an SQL statement
- Words that are reserved for specific uses when you are working on an SQL statement

### **Words that are reserved outside of an SQL statement**

Outside of SQL statements, the reserved words are as follows in any combination of upper- and lower-case letters:

- absolute, add, all, any, as
- bigInt, bin, bind, blob, boolean, by, byName, byPosition
- call, case, char, clob, close, const, continue, converse, current
- dataItem, dataTable, date, dbChar, decimal, decrement, delete, display
- else, embed, end, escape, execute, exit, extends, externallyDefined
- false, field, first, float, for, forEach, form, formGroup, forUpdate, forward, freeSql, from, function
- get, goto, group
- handler, hex, hold
- if, import, in, inOut, insert, int, interface, interval, into, is, isa



- label, languageBundle, last, library, like
- matches, mbChar, money, move
- new, next, nil, no, noRefresh, not, nullable, num, number, numc
- of, onEvent, onException, open, openUI, otherwise, out
- pacf, package, pageHandler, passing, prepare, previous, print, private, program
- record, ref, relative, replace, return, returning, returns
- scroll, self, service, set, show, singleRow, smallFloat, smallInt, sql, sqlCondition, stack, static, string
- this, time, timestamp, to, transaction, transfer, true, try, type
- unicode, update, url, use, using, usingKeys
- when, where, while, with, withinParent, wrap
- yes

## Words that are reserved in an SQL statement

In SQL statements, the reserved words are as follows in any combination of upper- and lower-case letters:

- absolute, action, add, alias, all, allocate, alter, and, any, are, as, asc, assertion, at, authorization, avg
- begin, between, bigint, binaryLargeObject, bit, bit\_length, blob, boolean, both, by
- call, cascade, cascaded, case, cast, catalog, char, char\_length, character, character\_length, characterLargeObject, characterVarying, charLargeObject, charVarying, check, clob, close, coalesce, collate, collation, column, comment, commit, connect, connection, constraint, constraints, continue, convert, copy, corresponding, count, create, cross, current, current\_date, current\_time, current\_timestamp, current\_user, cursor
- data, database, date, dateTime, day, deallocate, dec, decimal, declare, default, deferrable, deferred, delete, desc, describe, diagnostics, disconnect, distinct, domain, double, doublePrecision, drop
- else, end, endExec, escape, except, exception, exec, execute, exists, explain, external, extract
- false, fetch, first, float, for, foreign, found, from, full
- get, getCurrentConnection, global, go, goto, grant, group
- having, hour
- identity, image, immediate, in, index, indicator, initially, inner, input, insensitive, insert, int, integer, intersect, into, is, isolation
- join
- key
- language, last, leading, left, level, like, local, long, longint, lower, ltrim
- match, max, min, minute, module, month
- national, nationalCharacter, nationalCharacterLargeObject, nationalCharacterVarying, nationalCharLargeObject, nationalCharVarying, natural, nchar, ncharVarying, nclob, next, no, not, null, nullIf, number, numeric
- octet\_length, of, on, only, open, option, or, order, outer, output, overlaps
- pad, partial, position, prepare, preserve, primary, prior, privileges, procedure, public
- raw, read, real, references, relative, restrict, revoke, right, rollback, rows, rtrim, runtimeStatistics
- schema, scroll, second, section, select, session, session\_user, set, signal, size, smallint, some, space, sql, sqlcode, sqlerror, sqlstate, substr, substring, sum, system\_user
- table, tablespace, temporary, terminate, then, time, timestamp, timezone\_hour, timezone\_minute, tinyint, to, trailing, transaction, translate, translation, trim, true
- uncatalog, union, unique, unknown, update, upper, usage, user, using
- values, varbinary, varchar, varchar2, varying, view
- when, whenever, where, with, work, write
- year



EGL uses an import statement to resolve the name of a part. You specify a quoted string that has one or more directory names, each separated from the next by a semicolon.

**-generateFile** *genFile*

The absolute or relative path of the EGL file that contains the part you want to process. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

**-buildDescriptorFile** *bdFile*

The absolute or relative path of the build file that contains the build descriptor. Relative paths are relative to the directory in which you run the command.

Embed the path in double quotes.

**-buildDescriptorName** *bdName*

The name of a build descriptor part that guides generation. The build descriptor must be at the top level of an EGL build (.eglbld) file.

**-sqlID** *sqlID*

Sets the value of build descriptor option sqlID.

**-sqlPassword** *sqlPW*

Sets the value of build descriptor option sqlPassword.

**-destUserid** *destID*

Sets the value of build descriptor option destUserID.

**-destPassword** *destPW*

Sets the value of build descriptor option destPassword.

The eglpath value that you specify when invoking the command EGLSDK takes precedence over any eglpath value in an EGL command file. Similarly, build descriptor options that you specify when invoking the command take precedence over options in any build descriptor that is listed in an EGL command file.

## Examples

In the commands that follow, each multiline example belongs on a single line:

```
java EGLSDK "commandfile.xml"
```

```
java EGLSDK "commandfile.xml"  
-eglpath "c:\myGroup;h:\myCorp"
```

```
java EGLSDK generate  
-eglpath "c:\myGroup;h:\myCorp"  
-generateFile "c:\myProg.eglpgm"  
-buildDescriptorFile "c:\myBuild.eglbld"  
-buildDescriptorName myBuildDescriptor
```

```
java EGLSDK "myCommand.xml"  
-sqlID myID -sqlPassword myPW  
-destUserID myUserID -destPassword myPass
```

### Related concepts

“Build descriptor part” on page 383

“Generation from the EGL Software Development Kit (SDK)” on page 418

“Import” on page 33

“Master build descriptor” on page 386

### Related tasks

“Generating from the EGL Software Development Kit (SDK)” on page 418

### Related reference

“destPassword” on page 475

“destUserID” on page 475

“EGL build path and eglpath” on page 571

“sqlID” on page 491

“sqlPassword” on page 492

“Syntax diagram for EGL statements and commands” on page 884

## Format of eglmaster.properties file

The eglmaster.properties file is a Java properties file that the EGL SDK uses to specify the name and file path name of the master build descriptor. This properties file must be contained in a directory that is specified in the CLASSPATH variable of the process that invokes the EGLSDK command. The format of the eglmaster.properties file is as follows:

```
masterBuildDescriptorName=desc
masterBuildDescriptorFile=path
```

where:

*desc*

The name of the master build descriptor

*path*

The fully qualified path name of the EGL file in which the master build descriptor used by the EGL SDK is declared

The content of this file must follow the rules of a Java properties file. You can use either a slash (/) or two backslashes (\\) to separate file names within a path name.

You must specify both the **masterBuildDescriptorName** and **masterBuildDescriptorFile** keywords in the properties file. Otherwise the eglmaster.properties file is ignored.

Following is an example of the contents of an eglmaster.properties file:

```
# Specify the name of the master build descriptor:
masterBuildDescriptorName=MYBUILDDESRIPTOR
# Specify the file that contains the master build descriptor:
masterBuildDescriptorFile=d:/egl/builddescriptors/master.egl
```

### Related concepts

“Master build descriptor” on page 386

### Related tasks

“Choosing options for Java generation” on page 389

### Related reference

“Build descriptor options” on page 464

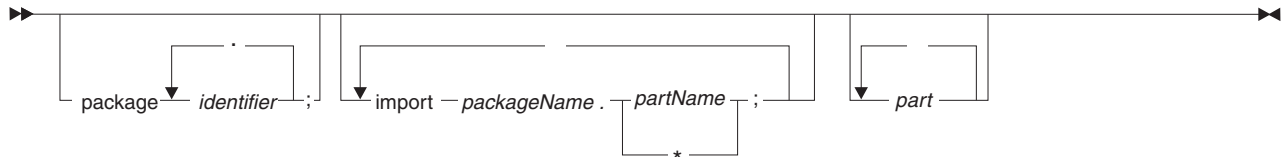
“EGLSDK” on page 583

“Format of master build descriptor plugin.xml file” on page 602

---

## EGL source format

You declare logic, data, and user-interface parts in EGL source files, each of which has the extension *.egl* and is constructed as follows:



### **package** *identifier*

Specifies the name of the package in which the file resides, with each identifier separated from the next by a period.

For an overview, see *EGL projects, packages, and files*.

### **import** *packageName*

Specifies the full name of a package to import. For an overview, see *Import*.

### *partName*

Specifies a single part to import.

- \* Indicates that every part in the package is to be imported.

### *part*

One of the EGL logic, data, or user-interface parts.

You may place comments in an EGL file, inside or outside a part.

### **Related concepts**

- "EGL projects, packages, and files" on page 15
- "Import" on page 33
- "References to parts" on page 23
- "Parts" on page 19

### **Related tasks**

- "Syntax diagram for EGL statements and commands" on page 884

### **Related reference**

- "Basic record part in EGL source format" on page 461
- "Comments" on page 531
- "DataItem part in EGL source format" on page 566
- "DataTable part in EGL source format" on page 568
- "FormGroup part in EGL source format" on page 603
- "Form part in EGL source format" on page 606
- "Function part in EGL source format" on page 621
- "Indexed record part in EGL source format" on page 632
- "Library part in EGL source format" on page 756
- "MQ record part in EGL source format" on page 769
- "PageHandler part in EGL source format" on page 785
- "Program part in EGL source format" on page 841
- "Relative record part in EGL source format" on page 865
- "Serial record part in EGL source format" on page 868
- "SQL record part in EGL source format" on page 877

---

## EGL system exceptions

The EGL system exceptions are available throughout your code, but are most often used in an **onException** block. For an overview, see *Exception handling*.

Each of the EGL system exceptions has at least the following fields:

### **code**

A string that identifies the exception; for example "com.ibm.egl.InvocationException" or the equivalent constant, SysLib.InvocationException

### **description**

A string that tells the meaning of the exception

The EGL system exceptions are as follows:

### **SysLib.DLIException**

Identifies a hard I/O error caused by DL/I access. Exception-specific fields are as follows:

#### **statusCode**

A 2-character DLI status code such as GB or II.

#### **pcbName**

The name of the PCB that was used in the DL/I call. In the program, in the variable of type PSBRecord, the same PCB name is referenced in the pcbName field of the complex property@PCB. The default value of that field is the name of the PCB record that was used on the call.

### **SysLib.FileIOException**

Identifies an error that occurs during file access. Errors that occur during relational-database or message queue access do not raise this exception. Exception-specific fields are as follows:

#### **errorCode**

The 8-character status code also returned in SysVar.ErrorCode; for details, see *SysVar.ErrorCode*

#### **fileName**

The logical name of the file being accessed; for details, see *Resource associations and file types*

### **SysLib.InvocationException**

Identifies an error that occurs in a **call** statement.

Exception-specific fields are as follows:

#### **errorCode**

The 8-character status code also returned in SysVar.ErrorCode; for details, see *SysVar.ErrorCode*

#### **name**

The name of the program being called.

### **SysLib.JavaObjectException**

Identifies an error that occurs during access of a Java method by way of an EGL interface. The **description** field contains the message from the Java exception. Exception-specific fields are as follows:

#### **exceptionName**

Name of the Java exception.

### **SysLib.LobProcessingException**

Identifies an error that occurred during processing of a field of type LOB or CLOB. Exception-specific fields are as follows:

#### **itemName**

Name of the field

#### **operation**

Name of the EGL system function that failed

#### **resource**

Name of the file (if any) attached to the field

### **SysLib.ServiceBindingException**

Identifies an error that occurs when initializing a binding in a services binding library or when invoking a ServiceLib function that changes a binding. No exception-specific fields are available.

### **SysLib.ServiceInvocationException**

Identifies an error that occurs when a service is invoked. An error might occur if an EGL or JAX-RPC class is missing; if the JAX-RPC runtime throws an exception; or if the EGL service (or EGL runtime) throws an exception.

The value of the **description** field varies by error type:

- Aside from EGL-specific exceptions, the value of the **diagnostic** field is from the toString method of the exception; for example, from JAX-RPC or SOAP
- If the error is from EGL, the value is the EGL Java runtime message number and message text, which are described in the topics that are subordinate to *Java runtime error codes*

In the case of the following EGL errors, only the **code** and **description** fields of the exception receive values:

- VGJ1501E: Error loading property file.
- VGJ1502E: Error loading service properties.
- VGJ1503E: Service binding error. The service is an EGL service and get/set Web Service properties are not valid.
- VGJ1504E: Service binding error. The service is a Web Service and get/set EGL service properties are not valid.
- VGJ1505E: Service binding error. The service is a Local EGL service and get/set TCP/IP service properties are not valid.

For other errors, the exception-specific fields are as follows:

#### **faultCode**

The value depends on the exception type:

- For SOAP fault exceptions, the value is returned from the faultCode of a SOAP exception
- For JAX-RPC exceptions, the value is blank
- For an EGL-related exception, the value is one of these message numbers: CSO7488E, CSO8109E, VGJ1525E, VGJ1526E, VGJ1527E, VGJ1528E, VGJ1529E, VGJ1530E, VGJ1532E, VGJ1534E, VGJ1535E, VGJ1536E, VGJ1538E, VGJ1539E, VGJ1540E, VGJ1541E, VGJ1542E, VGJ1543E, VGJ1544E, VGJ1545E

#### **source**

The type of service that was being invoked when the exception occurred:

- EGL, which indicates that an EGL service was being invoked

- WEB, which indicates that a Web service was being invoked

#### **location**

Location of the service at which the exception occurred:

- For EGL services accessed directly, the value is blank
- For EGL services accessed by TCP/IP, the value (if available) is formatted as follows:

`host:portNumber`

*host*

TCP/IP host name that refers to the machine where the service runs

*portNumber*

Number of the TCP/IP port that provides access to the service

- For Web services, the value is the URL; specifically, the SOAPActor value of the SOAP fault

#### **diagnostic**

Aside from EGL-specific exceptions, the value of the **diagnostic** field is from the `toString` method of the exception; for example, from JAX-RPC or SOAP. For EGL-specific exceptions, the value is blank.

### **SysLib.MQIOException**

Identifies an error that occurs during access of an MQSeries message queue. Exception-specific fields are as follows:

#### **errorCode**

The 8-character status code also returned in `SysVar.ErrorCode`; for details, see *SysVar.ErrorCode*

#### **mqConditionCode**

The completion code from an MQSeries API call, as described in *VGVar.mqConditionCode*

#### **name**

The logical name of the queue being accessed; for details, see *Resource associations and file types*

### **SysLib.SQLException**

Identifies an error that occurs during access of a relational database. Exception-specific fields are as follows:

#### **sqlca**

The SQL communication area; for details, see *SysVar.sqlca*

#### **sqlcode**

The SQL return code; for details, see *SysVar.sqlcode*

#### **sqlErrd**

A 6-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option; for details, see *VGVar.sqlErrd*

#### **sqlErrmc**

The error message associated with `sqlcode`, for database access other than through JDBC; for details, see *VGVar.sqlErrmc*

#### **sqlState**

The SQL state value for the most recently completed SQL I/O operation; for details, see *SysVar.sqlState*



### sqlWarn

An 11-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater than the warning number in the SQL SQLCA description; for details, see *VGVar.sqlState*

### Related concepts

"DL/I database support" on page 310 "EGL services and Web services" on page 158  
"Resource associations and file types" on page 393

### Related reference

"@DLI" on page 322  
"EGL Java runtime error codes" on page 1097  
"Exception handling" on page 94  
"errorCode" on page 1068  
"mqConditionCode" on page 1084  
"sqlca" on page 1071  
"sqlcode" on page 1072  
"sqlState" on page 1073  
"sqlerrd" on page 1085  
"sqlerrmc" on page 1086  
"sqlWarn" on page 1087

---

## EGL system limits

No EGL-defined limits are in effect for the number of parts or the number of hierarchical levels in an EGL file. The following limits apply, however:

- A program can use no more than 32767 variables and literals, including fields within variables.
- A call statement can have no more than 30 arguments; also, these restrictions apply to the size of the arguments in total—
  - Can be no more than 32567 if `remoteCall` or `ejbCall` is the value of the **type** property for the call.Both properties are in the linkage options part, `callLink` element.
- A field can be no more than 32767 bytes.
- In most cases, a numeric literal or field can have no more than 32 digits plus a sign, decimal point, or both; but a field that receives the result created by invoking the **mathLib.ROUND** function can be no more than 31 digits plus a sign, decimal point, or both.
- A static array can have no more than 7 dimensions and can have no more than 32767 elements in total.
- The situation for a dynamic array is as follows:
  - A dynamic array can have no more than 14 dimensions. The number of dimensions in a dynamic record array is one (for the record array declaration) plus the number of dimensions in the record structure.
  - A dynamic array can have a maximum size no greater than 2,147,483,647 elements. That number is in effect if you do not specify a maximum size, but the size that can be allocated is further limited by the memory available at run time.
  - The total size for all arguments that can be passed on a remote call is limited by the maximum buffer size supported for the protocol.

#### Related reference

"callLink element" on page 499

"round()" on page 980

"Naming conventions" on page 778

"parmForm in callLink element" on page 508

"type in callLink element" on page 515

---

## Expressions

An expression is a series of operands and operators that you specify when you write a program or function script.

Each expression resolves to a particular type of value at run time. A *numeric expression* resolves to a number; a *string expression* resolves to a series of characters; a *logical expression* resolves to true or false; a *datetime expression* resolves to a date, interval, time, or timestamp.

Expressions are evaluated in accordance with a set of precedence rules and (within a given level of precedence) from left to right, but you can use parentheses to force a different ordering. A nested parenthetical subexpression is evaluated before the enclosing parenthetical subexpression, and all parenthetical expressions are evaluated before the expression as a whole.

At a given level of evaluation, the first operand determines the type of expression (or subexpression). Consider this example:

```
"A value = " + 1 + 2
```

The first operand is of a character type, and the expression is a text expression with the following value:

```
"A value = 12"
```

Consider a different text expression:

```
"A value = " + (1 + 2)
```

The value in this case is as follows:

```
"A value = 3"
```

#### Related reference

"Datetime expressions"

"Logical expressions" on page 593

"Numeric expressions" on page 600

"Text expressions" on page 601

## Datetime expressions

A *datetime expression* resolves to a value of type DATE, INT, INTERVAL, TIME, or TIMESTAMP, depending on the context. A datetime expression must include one of these:

- A variable that contains a value of one of those types.
- A function invocation that returns a datetime value. Several system functions create a datetime value from a string literal or constant:
  - **DateTimeLib.dateValue** creates a date
  - **DateTimeLib.intervalValue** creates an interval
  - **DateTimeLib.timeValue** creates a time

- **DateTimeLib.timestampValue** creates a timestamp

Also, the system function **DateTimeLib.extend** returns a timestamp value that is longer or shorter than an input field of type DATE, TIME, or TIMESTAMP.

The next table summarizes the types of arithmetic operations that are valid in a datetime expression. As shown, a datetime expression may include a numeric expression that returns a number, but only in a subset of cases.

Arithmetic operations in a datetime expression

Type of Operand 1	Operator	Type of Operand 2	Type of Result	Comments
DATE	-	DATE	INT	
DATE	+/-	NUMBER	DATE	
NUMBER	+	DATE	DATE	
TIME STAMP	-	TIMESTAMP	INTERVAL	INTERVAL(dd, ss) unless Operand 1 and Operand 2 are both any of the following: <ul style="list-style-type: none"> <li>• TIMESTAMP(yyyy)</li> <li>• TIMESTAMP(yyyyMM)</li> <li>• TIMESTAMP(MM)</li> </ul> In those three cases, the result is INTERVAL(yyyyMM)
DATE	-	TIMESTAMP	INTERVAL	INTERVAL(ddssmmffffff)
TIME STAMP	-	DATE	INTERVAL	INTERVAL(ddHHmmssffffff)
TIME STAMP	+/-	INTERVAL	TIMESTAMP	
INTERVAL	+	TIMESTAMP	TIMESTAMP	
DATE	+/-	INTERVAL	TIMESTAMP	
INTERVAL	+	DATE	TIMESTAMP	
INTERVAL	+/-	INTERVAL	INTERVAL	Operand1 and Operand2 must both have (at most) years and months or both must have (at most) days and a time value
INTERVAL	*/	NUMBER	INTERVAL	

#### Related reference

“Assignments” on page 456  
 “dateValue()” on page 922  
 “extend()” on page 924  
 “intervalValue()” on page 924  
 “timeValue()” on page 928  
 “timestampValue()” on page 927  
 “Expressions” on page 591  
 “Logical expressions” on page 593  
 “Numeric expressions” on page 600  
 “Operators and precedence” on page 779

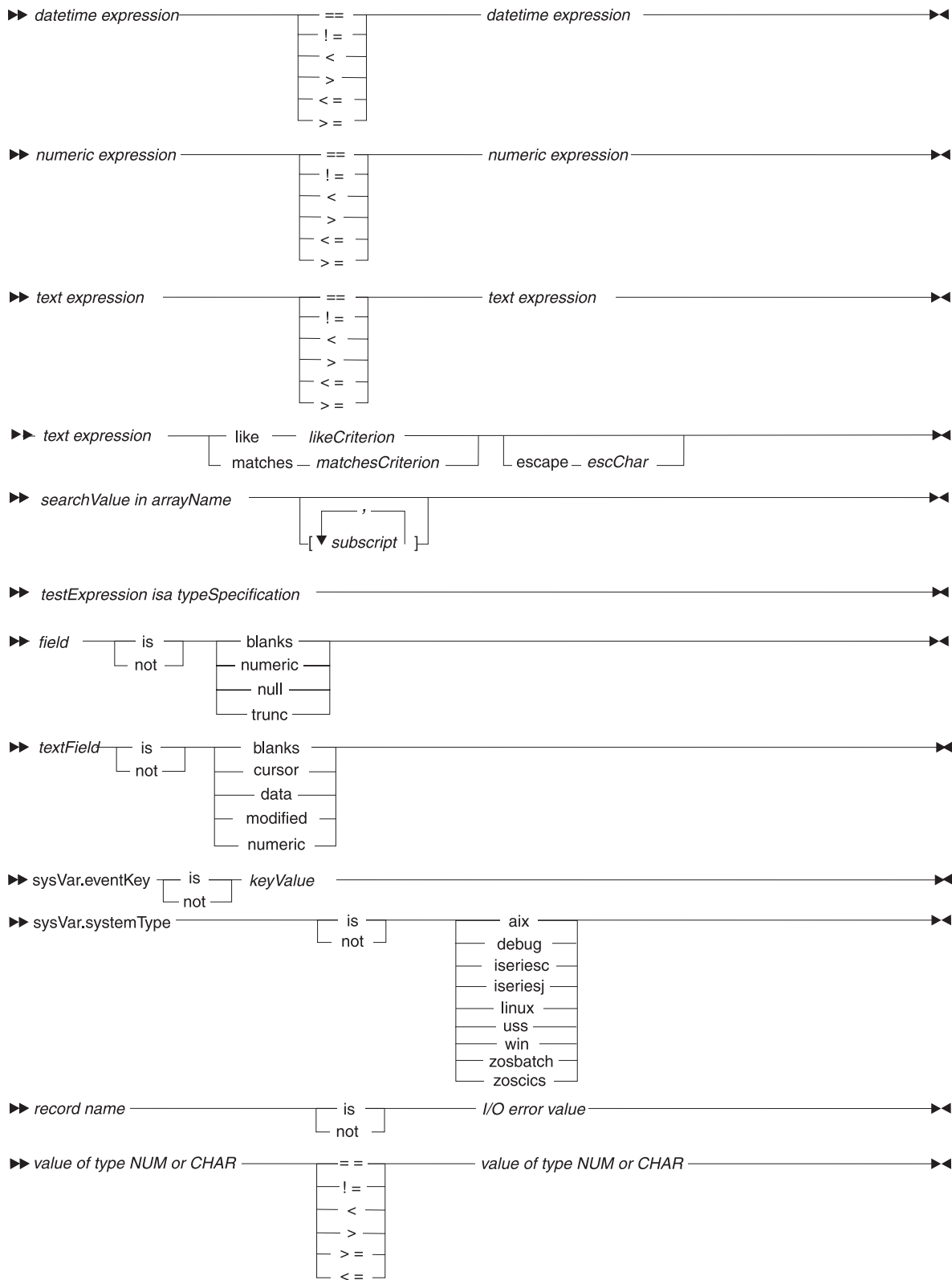
"Primitive types" on page 34  
"Text expressions" on page 601  
"Substrings" on page 882

## Logical expressions

A *logical expression* resolves to true or false and is used as a criterion in an **if** or **while** statement or (in some situations) in a **case** statement.

### Elementary logical expressions

An elementary logical expression is composed of an operand, a comparison operator, and a second operand, as shown in this syntax diagram and the subsequent table:



First operand	Comparison Operator	Second operand
<i>datetime expression</i>	One of these: ==, !=, <, >, <=, >=	<i>datetime expression</i>  The first and second expressions must be of compatible types.  In the case of datetime comparisons, the greater than sign (>) means later in time; and the less than (<) sign means earlier in time.
<i>numeric expression</i>	One of these: ==, !=, <, >, <=, >=	<i>numeric expression</i>
<i>string expression</i>	One of these: ==, !=, <, >, <=, >=	<i>string expression</i>
<i>string expression</i>	like	<i>likeCriterion</i> , which is a character field or literal against which <i>string expression</i> is compared, character position by character position from left to right. Use of this feature is similar to the use of keyword <b>like</b> in SQL queries.  <i>escChar</i> is a one-character field or literal that resolves to an escape character.  For further details, see <i>like operator</i> .
<i>string expression</i>	matches	<i>matchCriterion</i> , which is a character field or literal against which <i>string expression</i> is compared, character position by character position from left to right. Use of this feature is similar to the use of <i>regular expressions</i> in UNIX or Perl.  <i>escChar</i> is a one-character field or literal that resolves to an escape character.  For further details, see <i>matches operator</i> .
Value of type NUM or CHAR, as described for the second operand	One of these: ==, !=, <, >, <=, >=	Value of type NUM or CHAR, which can be any of these: <ul style="list-style-type: none"> <li>• A field that is of type NUM and has no decimal places</li> <li>• An integer literal</li> <li>• A field or literal of type CHAR</li> </ul>
<i>searchValue</i>	in	<i>arrayName</i> ; for details, see <i>in</i> .
<i>field not in SQL record</i>	One of these: <ul style="list-style-type: none"> <li>• is</li> <li>• not</li> </ul>	One of these: <ul style="list-style-type: none"> <li>• blanks (for testing whether the value of a character field is or is not blanks only)</li> <li>• numeric (for testing whether the value of a field of type CHAR or MBCHAR is or is not numeric)</li> </ul>

First operand	Comparison Operator	Second operand
<i>field in an SQL record</i>	One of these: <ul style="list-style-type: none"> <li>• is</li> <li>• not</li> </ul>	One of these: <ul style="list-style-type: none"> <li>• blanks (for testing whether the value of a character field is or is not blanks only)</li> <li>• null (for testing whether the field was set to null either by a set statement or by reading from a relational database)</li> <li>• numeric (for testing whether the value of a field of type CHAR or MBCHAR is or is not numeric)</li> <li>• trunc (for testing whether non-blank characters were deleted on the right when a single- or double-byte character value was last read from a relational database into the field)</li> </ul> <p>The trunc test can resolve to true only when the database column is longer than the field. The value for the test is false after a value is moved to the field or after the field is set to null.</p>
<i>textField</i> (the name of a field in a text form)	One of these: <ul style="list-style-type: none"> <li>• is</li> <li>• not</li> </ul>	One of these: <ul style="list-style-type: none"> <li>• blanks (for testing whether the value of the text field is or is not limited to blanks or nulls).</li> </ul> <p>The test for blanks is based on the user's last input to the form, not on the current contents of the form field; and a test that uses <i>is</i> is true in these cases:</p> <ul style="list-style-type: none"> <li>– The user's last input was blanks or null; or</li> <li>– The user entered no data in the field since the start of the program or since a set statement ran that was of type <i>set form initial</i>.</li> </ul> <ul style="list-style-type: none"> <li>• cursor (for testing whether the user left the cursor in the specified text field).</li> <li>• data (for testing whether data other than blanks or nulls is in the specified text field).</li> <li>• modified (for testing whether the field's modified data tag is set, as described in Modified data tag and property).</li> <li>• numeric (for testing whether the value of a field of type CHAR or MBCHAR is or is not numeric).</li> </ul>
<i>ConverseVar.eventKey</i>	One of these: <ul style="list-style-type: none"> <li>• is</li> <li>• not</li> </ul>	For details, see <i>ConverseVar.eventKey</i> .
<i>sysVar.systemType</i>	One of these: <ul style="list-style-type: none"> <li>• is</li> <li>• not</li> </ul>	For details, see <i>sysVar.systemType</i> .  You cannot use <i>is</i> or <i>not</i> to test a value returned by <i>VGLib.getVAGSysType</i> .

First operand	Comparison Operator	Second operand
<i>record name</i>	One of these: <ul style="list-style-type: none"> <li>• is</li> <li>• not</li> </ul>	An I/O error value appropriate for the record organization. See <i>I/O error values</i> .

The next table lists the comparison operators, each of which is used in an expression that resolves to true or false.

Operator	Purpose
==	The <i>equality</i> operator indicates whether two operands have the same value.
!=	The <i>not equal</i> operator indicates whether two operands have different values.
<	The <i>less than</i> operator indicates whether the first of two operands is numerically less than the second.
>	The <i>greater than</i> operator indicates whether the first of two operands is numerically greater than the second.
<=	The <i>less than or equal to</i> operator indicates whether the first of two operands is numerically less than or equal to the second.
>=	The <i>greater than or equal to</i> operator indicates whether the first of two operands is numerically greater than or equal to the second.
in	The <i>in</i> operator indicates whether the first of two operands is a value in the second operand, which references an array. For details, see <i>in</i> .
is	The <i>is</i> operator indicates whether the first of two operands is in the category of the second. For details, see the previous table.
like	The <i>like</i> operator indicates whether the characters in the first of two operands is matched by the second operand, as described in <i>like operator</i> .
matches	The <i>matches</i> operator indicates whether the characters in the first of two operands is matched by the second operand, as described in <i>matches operator</i> .
not	The <i>not</i> operator indicates whether the first of two operands is not in the category of the second. For details, see the previous table.

The next table and the explanations that follow tell the compatibility rules when the operands are of the specified types.

Primitive type of first operand	Primitive type of second operand
BIN	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACE, SMALLFLOAT
CHAR	CHAR, DATE, HEX, MBCHAR, NUM, TIME, TIMESTAMP
DATE	CHAR, DATE, NUM, TIMESTAMP
DBCHAR	DBCHAR
DECIMAL	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACE, SMALLFLOAT
HEX	CHAR, HEX
MBCHAR	CHAR, MBCHAR



Primitive type of first operand	Primitive type of second operand
MONEY	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
NUM	BIN, CHAR, DATE, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT, TIME
NUMC	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
PACF	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
TIME	CHAR, NUM, TIME, TIMESTAMP
TIMESTAMP	CHAR, DATE, TIME, TIMESTAMP
UNICODE	UNICODE

Details are as follows:

- A value of any of the numeric types (BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT) can be compared to a value of any numeric type and size, and EGL does temporary conversions as appropriate. An equality comparison of equivalent fractions (like 1.4 and 1.40) evaluates to true, even if the decimal places are different.
- A value of type CHAR can be compared to a value of type HEX only if each character of type CHAR is within the range of hexadecimal digits (0-9, A-F, a-f). EGL temporarily converts any lowercase letters to uppercase in the value of type CHAR.
- If a comparison includes two values of a fixed character type (CHAR, DBCHAR, HEX, MBCHAR, UNICODE, limited-length STRING) and one value has fewer bytes than the other, a temporary conversion pads the shorter value on the right:
  - In a comparison with a value of type MBCHAR, a value of type CHAR or limited-length STRING is padded on the right with single-byte blanks
  - In a comparison with a value of type HEX, a value of type CHAR or limited-length STRING is padded on the right with binary zeros
  - A value of type DBCHAR is padded on the right with double-byte blanks
  - A value of type UNICODE is padded on the right with Unicode double-byte blanks
  - A value of type HEX is padded on the right with binary zeros, which means (for example) that if a value "0A" must be expanded to two bytes, the value for comparison purposes is "0A00" rather than "000A"
- Any trailing blanks in a limited-length string are ignored during the comparison itself, after any padding occurs. Trailing blanks in other fields of type STRING, however, are not ignored.
- A value of type CHAR can be compared to a value of type NUM only if these conditions apply:
  - The value of type CHAR has single-byte digits, with no other characters
  - The definition of the value of type NUM has no decimal point

A CHAR-to-NUM comparison works as follows:

- A temporary conversion puts the NUM value into a CHAR format. The numeric characters are left-justified, with additional single-byte blanks as needed. If a NUM-type field of length 4 has a value of 7, for example, the value is treated as "7" with three blanks on the right.

- If the length of the fields do not match, a temporary conversion pads the shorter value with blanks on the right.
- The comparison checks the values byte-by-byte. Consider two examples:
  - A CHAR-type field of length 2 and value "7 " (including a blank) is equal to a NUM-type field of length 1 and value 7 because the temporary field that is based on the NUM-type field also includes a final blank
  - A CHAR-type field of value "8" is greater than a NUM-type field of value of 534 because the "8" comes after "5" in the ASCII or EBCDIC search order

## Complex logical expressions

You can build a more complex expression by using either an *and* (&&) or *or* operator (||) to combine a pair of more elementary expressions. In addition, you can use the *not* operator (!), as described later.

If a logical expression is composed of elementary logical expressions that are combined by *or* operators, EGL evaluates the expression in accordance with the rules of precedence, but stops the evaluation if one of the elementary logical expressions resolves to true. Consider an example:

```
field01 == field02 || 3 in array03 || x == y
```

If field01 does not equal field02, evaluation proceeds. If the value 3 is in array03, however, the overall expression is proven to be true, and the last elementary logical expression (x == y) is not evaluated.

Similarly, if elementary logical expressions are combined by *and* operators, EGL stops the evaluation if one of the elementary logical expressions resolves to false. In the following example, evaluation stops as soon as field01 is found to be unequal to field02:

```
field01 == field02 && 3 in array03 && x == y
```

You may use paired parentheses in a logical expression for any of these purposes:

- To change the order of evaluation.
- To clarify your meaning.
- To make possible the use of the *not* operator (!), which resolves to a Boolean value (true or false) opposite to the value of a logical expression that immediately follows. The subsequent expression must be in parentheses.

## Examples

In reviewing the examples that follow, assume that value1 contains "1", value2 contains "2", and so on:

```
/* == true */
value5 < value2 + value4

/* == false */
!(value1 is numeric)

/* == true when the generated output runs
   on Windows 2000, Windows NT,
   or z/OS UNIX System Services */
sysVar.systemType is WIN || sysVar.systemType is USS

/* == true */
(value6 < 5 || value2 + 3 >= value5) && value2 == 2
```

## Related concepts

“Modified data tag and modified property” on page 191

## Related tasks

“Syntax diagram for EGL statements and commands” on page 884

## Related reference

“case” on page 668  
“Datetime expressions” on page 591  
“eventKey” on page 1058  
“Exception handling” on page 94  
“Expressions” on page 591  
“getVAGSysType()” on page 1054  
“I/O error values” on page 638  
“if, else” on page 715  
“in operator” on page 629  
“like operator” on page 763  
“like operator” on page 763  
“Numeric expressions”  
“Operators and precedence” on page 779  
“Primitive types” on page 34  
“Text expressions” on page 601  
“STRING” on page 40  
“systemType” on page 1074  
“while” on page 755

## Numeric expressions

A *numeric expression* resolves to a number, and you specify such an expression in various situations; for example, on the right side of an assignment statement. A numeric expression may be composed of any of these:

- A numeric operand, which is one of these:
  - A variable that contains a number. The item may be preceded with a sign.
  - A numeric literal, which may begin with a sign, but always has a series of digits and may include a single decimal point.
  - A function invocation that returns a number.

The type of a numeric literal is implied by the value of that literal:

- An integer of 4 digits or less is of type SMALLINT
  - An integer of 5 to 8 digits is of type INT
  - An integer of 9 to 18 digits is of type BIGINT
  - A number that includes a decimal point is of type NUM
- A numeric operand, followed by a numeric operator, followed by a second numeric operand.
  - A more complex expression formed by using a numeric operator to combine a pair of more elementary expressions.

You may use paired parentheses in a numeric expression to change the order of evaluation or to clarify your meaning.

In reviewing the examples that follow, assume that intValue1 equals 1, intValue2 equals 2, and so on, and that each value has no decimal places:

```

/* == -8, with the parentheses overriding
   the usual precedence of * and + */
intValue2 * (intValue1 - 5)

/* == -2, with a unary minus as the last operator */
intValue2 + -4

/* == 1.4, if the expression is assigned to an
   item with at least one decimal place. */
intValue7 / intValue5

/* == 2, which is a remainder
   expressed as an integer value */
intValue7 % intValue5

```

For an example that shows the effect of parentheses on the use of a plus (+) sign, see *Expressions*.

A numeric expression may give an unexpected result if an intermediate, calculated value requires more than 128 bits.

#### Related reference

"Datetime expressions" on page 591  
 "Expressions" on page 591  
 "Logical expressions" on page 593  
 "Operators and precedence" on page 779  
 "Primitive types" on page 34  
 "Text expressions"

## Text expressions

A *text expression* resolves to a series of characters, and you specify such an expression in various situations; for example, on the right side of an assignment statement. The text expression may be any of these:

- A variable that contains a series of characters.
- A *string literal*, which is a series of characters delimited by double quote marks. The literal is of type `STRING`.
- A substring of a literal or variable that contains a series of characters. For details, see *Substrings*.
- The invocation of any string-formatting system word that returns a series of characters. For details, see *String formatting (system words)*.
- A series of values of the previous kinds, where each value is separated from the next by the concatenation operator, which is a plus sign (+). The following statement assigns *WebSphere* to `myString`:

```
myString = "Web" + "Sphere";
```

For an example that shows the effect of parentheses on the use of a plus (+) sign, see *Expressions*.

- Any other function invocation that returns a series of characters.

Any character preceded with the escape character (\) is included in the text expression. In particular, you can use the escape character to include the following characters in a literal, field, or return value:

- A double quote mark (")
- A backslash (\)
- A backspace, as indicated by `\b`
- A form feed, as indicated by `\f`

- A newline character, as indicated by `\n`
- A carriage return, as indicated by `\r`
- A tab, as indicated by `\t`

Examples are as follows:

```
myString = "He said, \"Escape while you can!\"";
myString2 = "Is a backslash (\\) needed?";
```

An error occurs if a literal has no ending quote mark:

```
myString3 = "Escape is impossible\";
```

Each value in the text expression must be valid for the context in which the expression is used. For example, an item of type UNICODE cannot be used in an expression assigned to an item of type CHAR. Additional details are in *Assignments*.

#### Related reference

“Assignments” on page 456  
 “Datetime expressions” on page 591  
 “Expressions” on page 591  
 “Logical expressions” on page 593  
 “Numeric expressions” on page 600  
 “Operators and precedence” on page 779  
 “Primitive types” on page 34  
  
 “Substrings” on page 882

---

## Format of master build descriptor plugin.xml file

The master build descriptor plugin.xml file is an XML file that the workbench uses to specify the name and file path name of the master build descriptor. You need this only if you need a master build descriptor to enforce certain options to be used for generation and you are generating from the workbench or are using the EGLCMD command. You must put this plugin.xml file in a directory in the plugins directory. The format of the file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="id"
  name="plg"
  version="5.0"
  vendor-name="com">
  <requires />
  <runtime />
  <extension point =
    "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor">
    <masterBuildDescriptor file = "bfil" name = "mas" />
  </extension>
</plugin>
```

where:

*id* The identifier for the plug-in  
*plg* The name of the plug-in  
*com* The name of your company

*bfil*

The path name of a file containing a master build descriptor, of the form *project/folder/file*, relative to Enterprise Developer's workspace directory, where:

*project*

The name of the project directory

*folder*

The name of a directory within the project directory

*file* The name of a file that contains a master build descriptor

*mas*

The name of a master build descriptor

The content of this file must follow the rules of an XML file. To separate file names within a path name you must use the slash (/) character.

You must specify both the name attribute and the file attribute. Otherwise the plugin.xml file is ignored.

Following is an example of the contents of plugin.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example master BuildDescriptor Plugin -->

<plugin
  id="example.master.BuildDescriptor.plugin"
  name="Example master BuildDescriptor plug-in"
  version="5.0"
  vendor-name="IBM">
  <requires />
  <runtime />
  <!-- ===== -->
  <!-- Register the master BuildDescriptor -->
  <!-- ===== -->
  <extension point =
    "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor" >
    <masterBuildDescriptor file
      = "myProject/myFolder/myFile.eglbld" name = "masterBD" />
    </extension>
</plugin>
```

### Related concepts

"Build descriptor part" on page 383

"Master build descriptor" on page 386

### Related tasks

"Generating from the workbench batch interface" on page 417

"Generating in the workbench" on page 414

### Related reference

"Build descriptor options" on page 464

"EGLCMD" on page 572

"Format of eglmaster.properties file" on page 585

---

## FormGroup part in EGL source format

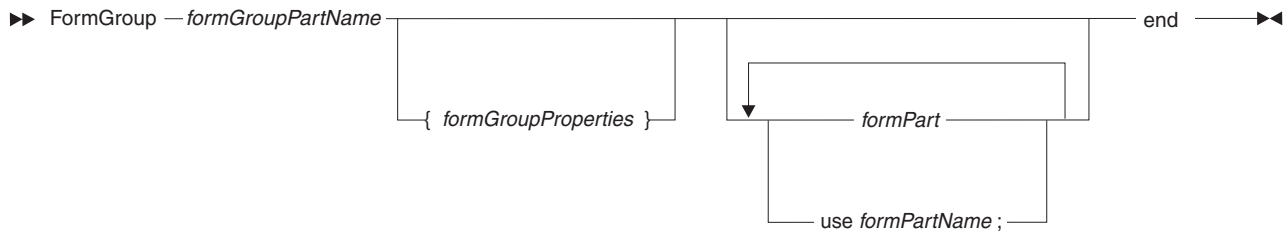
You declare a formGroup part in an EGL file, which is described in *EGL source format*. This part is a generatable part, which means that it must be at the top level of the file and must have the same name as the file.

A program can only use forms that are associated with a form group referenced by the program's use declaration.

An example of a formGroup part is as follows:

```
FormGroup myFormGroup
{
  validationBypassKeys = [pf3],
  helpKey = "pf1",
  pfKeyEquate = yes,
  screenFloatingArea
  {
    screenSize = [24,80],
    topMargin = 0,
    bottomMargin = 0,
    leftMargin = 0,
    rightMargin = 0
  },
  printFloatingArea
  {
    pageSize = [60,80],
    topMargin = 3,
    bottomMargin = 3,
    leftMargin = 5,
    rightMargin = 5
  }
}
use myForm01;
use myForm02;
end
```

The diagram of a formGroup part is as follows:



### **FormGroup** *formGroupPartName ... end*

Identifies the part as a form group and specifies the part name. For the rules of naming, see *Naming conventions*.

### *formGroupProperties*

A series of properties, each separated from the next by a comma. Each property is described later.

### *formPart*

A text or print form, as described in *Form part in EGL source format*.

### **use** *formPartName*

A use declaration that provides access to a form that is not embedded in the form group.

The form group properties are as follows:

### **alias**

A string that is incorporated into the names of generated output. If you do not specify an alias, the formGroup-part name is used instead.

**validationBypassKeys** = [*bypassKeyValue*]

Identifies one or more user keystrokes that causes the EGL runtime to skip input-field validations. This property is useful for reserving a keystroke that ends the program quickly. Each *bypassKeyValue* option is as follows:

**pf*n***

The name of an F or PF key, including a number between 1 and 24, inclusive.

**Note:** Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

If you wish to specify more than one key value, delimit the set of values with brackets and separate each value from the next with a comma, as in the following example:

```
validationBypassKeys = [pf3, pf4]
```

**helpKey** = "*helpKeyValue*"

Identifies a user keystroke that causes the EGL runtime to present a help form to the user. The *helpKeyValue* option is as follows:

**pf*n***

The name of an f or pf key, including a number between 1 and 24, inclusive.

**Note:** Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

**pfKeyEquate** = yes, **pfKeyEquate** = no

Specifies whether the keystroke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12. For details, see *pfKeyEquate*.

**screenFloatingArea** { *properties* }

Defines the floating area used for output to a screen. For an overview of floating areas, see *Form part*. For property details, see the next section.

**printFloatingArea** { *properties* }

Defines the floating area used for printable output. For an overview of floating areas, see *Form part*. For property details, see *Properties of a print floating area*.

## Properties of a screen floating area

The set of properties after **screenFloatingArea** is delimited by braces ({ }), and each property is separated from the next by a comma. The properties are as follows:

**screenSize** = [*rows*, *columns*]

Number of rows and columns in the online presentation area, including any lines or columns used as margins. The default is as follows:

```
screenSize=[24,80]
```

**topMargin**= *rows*

Number of lines left blank at the top of the presentation area. The default is 0.

**bottomMargin**= *rows*

Number of lines left blank at the bottom of the presentation area. The default is 0.



**leftMargin=** *columns*

Number of columns left blank at the left of the presentation area. The default is 0.

**rightMargin=** *columns*

Number of columns left blank at the right of the presentation area. The default is 0.

## Properties of a print floating area

The set of properties after **printFloatingArea** is delimited by braces ({ }), and each property is separated from the next by a comma. The properties are as follows:

**pageSize =** [*rows, columns*]

Number of rows and columns in the printable presentation area, including any lines or columns used as margins. This property is required if you specify a print floating area.

**deviceType = singleByte, deviceType = doubleByte**

Specifies whether the floating-area declaration is for a printer that supports single-byte output (as is the default) or double-byte output. Specify **doubleByte** if any of the forms include items of type DBCHAR or MBCHAR.

**topMargin =** *rows*

Number of lines left blank at the top of the presentation area. The default is 0.

**bottomMargin =** *rows*

Number of lines left blank at the bottom of the presentation area. The default is 0.

**leftMargin =** *columns*

Number of columns left blank at the left of the presentation area. The default is 0.

**rightMargin =** *columns*

Number of columns left blank at the right of the presentation area. The default is 0.

### Related concepts

"EGL projects, packages, and files" on page 15

"FormGroup part" on page 183

"Form part" on page 184

### Related reference

"EGL source format" on page 586

"Form part in EGL source format"

"Naming conventions" on page 778

"pfKeyEquate" on page 792

"Use declaration" on page 1091

---

## Form part in EGL source format

You declare a form part in an EGL file, which is described in *EGL source format*. If a form part is accessed by only one form group, it is recommended that the form part be embedded in the formGroup part. If a form part is accessed by multiple form groups, it is necessary to specify the form part at the top level of an EGL file.

An example of a text form is as follows:

```

Form myTextForm type textForm
{
    formsize= [24, 80],
    position= [1, 1],
    validationBypassKeys=[pf3, pf4],
    helpKey="pf1",
    helpForm="myHelpForm",
    msgField="myMsg",
    alias = "form1"
}

* { position=[1, 31], value="Sample Menu" } ;
* { position=[3, 18], value="Activity:" } ;
* { position=[3, 61], value="Command Code:" } ;

activity char(42)[5] { position=[4,18], protect=skip } ;

commandCode char(10)[5] { position=[4,61], protect=skip } ;

* { position=[10, 1], value="Response:" } ;
response char(228) { position=[10, 12], protect=skip } ;

* { position=[13, 1], value="Command:" } ;
myCommand char(70) { position=[13,10] } ;

* { position=[14, 1], value="Enter=Run F3=Exit" } ;

myMsg char(70) { position=[20,4] };

end

```

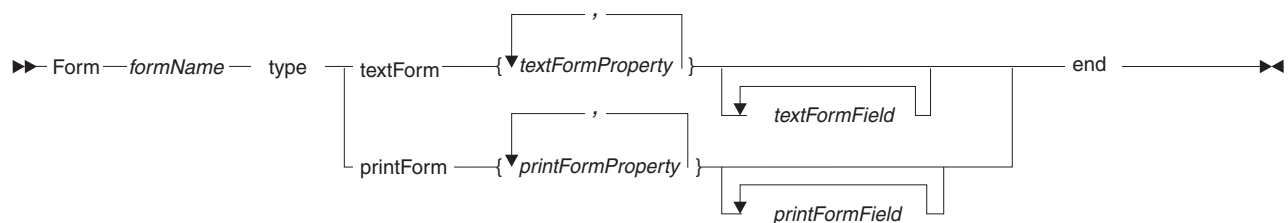
An example of a print form is as follows:

```

Form myPrintForm type printForm
{
    formsize= [48, 80],
    position= [1, 1],
    msgField="myMsg",
    alias = "form2"
}
* { position=[1, 10], value="Your ID: " } ;
ID char(70) { position=[1, 30] };
myMsg char(70) { position=[20, 4] };
end

```

The diagram of a form part is as follows:



### **Form *formName* ... end**

Identifies the part as a form and specifies the part name. For the rules of naming, see Naming conventions.

### **textForm**

Indicates that the form is a text form.

### *textFormProperty*

A text-form property. For details, see *Text form*.

*textFormField*

A text-form field. For details, see *Form fields*.

**printForm**

Indicates that the form is a print form.

*printFormProperty*

A print-form property. For details, see *Print form*.

*printFormField*

A print-form field. For details, see *Form fields*.

## Text-form properties

The text-form properties are as follows:

**formSize** = [*rows*, *columns*]

Number of rows and columns in the online presentation area. This property is required.

The column value is equivalent to the number of single-byte characters that can be displayed across the presentation area.

**position** = [*row*, *column*]

Row and column at which the form is displayed in the presentation area. If you omit this property, the form is a floating form and is displayed in the floating area, at the next free line where the entire form can fit in the floating area.

**validationBypassKeys** = [*bypassKeyValue*]

Identifies one or more user keystrokes that causes the EGL runtime to skip input-field validations. This property is useful for reserving a keystroke that ends the program quickly. The *bypassKeyValue* option is as follows:

**pf*n***

The name of an F or PF key, including a number between 1 and 24, inclusive

**Note:** Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

If you wish to specify more than one key value, delimit the set of values with parentheses and separate each value from the next with a comma, as in the following example:

```
validationBypassKeys = [pf3, pf4]
```

**helpKey** = "*helpKeyValue*"

Identifies a user keystroke that causes the EGL runtime to present a help form to the user. The *helpKeyValue* option is as follows:

**pf*n***

The name of an F or PF key, including a number between 1 and 24, inclusive

**Note:** Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

**helpForm** = "*formName*"

Name of the help form that is specific to the text form.

**msgField** = *"fieldName"*

Name of the text-form field that displays a message in response to a validation error or in response to the running of `ConverseLib.displayMsgNum`.

**alias** = *"alias"*

An alias of 8 characters or less, for use by the EGL runtime.

## Print-form properties

The print-form properties are as follows:

**formsize** = [*rows, columns*]

Number of rows and columns in the online presentation area. This property is required.

The column value is equivalent to the number of single-byte characters that can be displayed across the presentation area.

**position** = [*row, column*]

Row and column at which the form is displayed in the presentation area. If you omit this property, the form is a floating form and is displayed in the floating area, at the next free line where the entire form can fit in the floating area.

**msgField** = *"fieldName"*

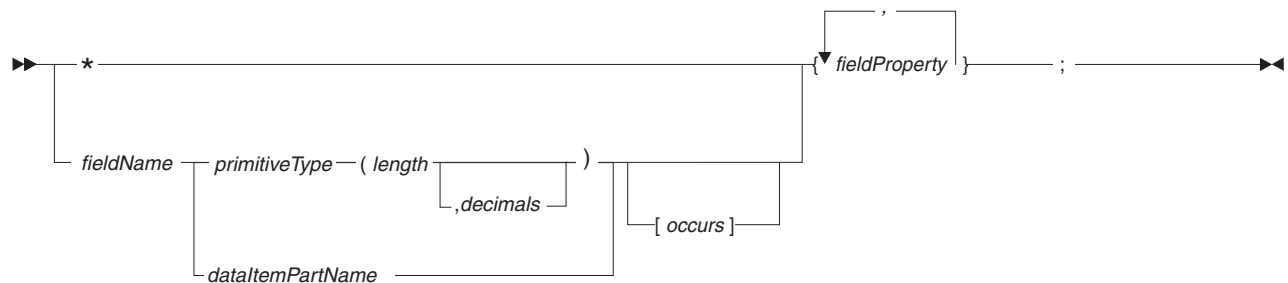
Name of the text-form field that displays a message in response to the running of `ConverseLib.displayMsgNum`.

**alias** = *"alias"*

An alias of 8 characters or less, for use by the EGL runtime.

## Form fields

The diagram of a form field is as follows:



- \* Indicates that the field is a constant field. It has no name but has a constant value, which is specified in the field-specific **value** property. Statements in your code cannot access the value in a constant field.

*fieldProperty*

A text-form field property. For details, see *Text-form field properties*.

*fieldName*

Specifies the name of the field. For rules, see *Naming conventions*.

Your code can access the value of a named field, which is also called a *variable field*.

If a text form contains a variable field that starts on one line and ends on another, the text form can be displayed only on screens where the screen width equals the width of the form.

*occurs*

The number of elements in a field array. Only one-dimensional arrays are supported. For further details, see *For field arrays*.

*primitiveType*

The primitive type assigned to the field. This specification affects the maximum length; but any numeric field is generated as type NUM.

Forms that contain fields of type DBCHAR can only be used on systems and devices that support double-byte character sets. Similarly, forms that contain fields of type MBCHAR can only be used on systems and devices that support multiple-byte character sets.

The primitive types FLOAT, SMALLFLOAT, and UNICODE are not supported for text or print forms.

*length*

The field's length, which is an integer that represents the maximum number of characters or digits that can be placed in the field.

*decimals*

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

*dataItemPartName*

The name of a dataItem part that is a model of format for the field, as described in *intypeDef*. The dataItem part must be visible to the form part, as described in *References to parts*.

## Text-form field properties

Properties that are useful only in text-form fields are described later. The following properties are used more widely and also available:

- "align" on page 801
- "currency" on page 804
- "currencySymbol" on page 805
- "dateFormat" on page 805
- "fillCharacter" on page 810
- "isBoolean" on page 812
- "lineWrap" on page 815
- "lowerCase" on page 815
- "masked" on page 816
- "numericSeparator" on page 820
- "outline" on page 820
- "sign" on page 825
- "timeFormat" on page 827
- "timestampFormat" on page 828
- "upperCase" on page 831
- "zeroFormat" on page 837

### For any field

The following properties are useful for any field on a form:

**position** = [row, column]

Row and column of the attribute byte that precedes the field. This property is required.

**value** = "stringLiteral"

A character string that is displayed in the field. Quotes are required.

This property can be specified for any item; for example, in a dataItem part declaration.

**Note:** If VisualAge Generator compatibility is in effect and you set the text-form property **value**, the content of that property is available in the program only after the user has returned the form. For this reason, the value that you set in the program does not need to be valid for the item in the program.

**fieldLen** = lengthInBytes

Field length; the number of single-byte characters that can be displayed in the field. This value does not include the preceding attribute byte.

The value of **fieldLen** for numeric fields must be great enough to display the largest number that can be held in the field, plus (if the number has decimal places) a decimal point. The value of **fieldLen** for a field of type CHAR, DBCHAR, MBCHAR, or UNICODE must be large enough to account for the double-byte characters, as well as any shift-in/shift-out characters.

The default **fieldLen** is the number of bytes needed to display the largest number possible for the primitive type, including all formatting characters.

## For variable text fields

The following properties are useful for variable text fields:

**cursor** = no, **cursor** = yes

Indicates whether the on-screen cursor is at the beginning of the field when the form is first displayed. Only one field in the form can have the cursor property set to *yes*. The default is *no*.

**modified** = no, **modified** = yes

Indicates whether the program will consider the field to have been modified, regardless of whether the user changed the value. For details, see *Modified data tag and modified property*.

The default is *no*.

**protect** = no, **protect** = skip, **protect** = yes

Specifies whether the user can access the field. Valid values are as follows:

**no (the default for variable fields)**

Sets the field so that the user can overwrite the value in it.

**skip (the default for constant fields)**

Sets the field so that the user cannot overwrite the value in it. In addition, the cursor skips the field in either of these cases:

- The user is working on the previous field in the tab order and either presses **Tab** or fills that previous field with content; or
- The user is working on the next field in the tab order and presses **Shift Tab**.

**yes**

Sets the field so that the user cannot overwrite the value in it.

**validationOrder** = *integer*

Indicates the field's position in the validation order. The default order in which the fields are validated is the order of the fields on screen, left to right, top to bottom.

## For field arrays

One-dimensional arrays are supported on text and print forms. In an array declaration, the value of the **occurs** property is greater than 1, as in this example:

```
myArray char(1)[3];
```

Array elements are positioned in relation to the placement specified for the first element in the array. The default behavior is to position the elements vertically on consecutive rows.

Use the following properties to vary the default behavior:

**columns** = *numberOfElements*

Number of array elements in each row. The default is 1.

**linesBetweenRows** = *numberOfLines*

Number of lines between each row that contains array elements. The default is 0.

**spacesBetweenColumns** = *numberOfSpaces*

Number of spaces between each array element. The default is 1.

**indexOrientation** = **down**, **indexOrientation** = **across**

Specifies how the program references the elements of an array:

- If you set **indexOrientation** to *down*, elements are numbered from top to bottom, then left to right, so that the elements in a given column are numbered sequentially. The value of **indexOrientation** is *down* by default.
- If you set **indexOrientation** to *across*, elements are numbered from left to right, then top to bottom, so that the elements in a given row are numbered sequentially.

You can override properties for an array element. In the following field declaration, for example, the **cursor** property is overridden in the second element of myArray:

```
myArray char(10)[5]
  {position=[4,61], protect=skip, myArray[2] { cursor = yes} };
```

## Related concepts

"Modified data tag and modified property" on page 191

"Overview of EGL properties" on page 64

"Print forms" on page 186

"References to parts" on page 23

"Text forms" on page 188

"Typedef" on page 28

## Related reference

"Field-presentation properties" on page 67

"Formatting properties" on page 67

"Naming conventions" on page 778

"NUM" on page 51

"Primitive types" on page 34

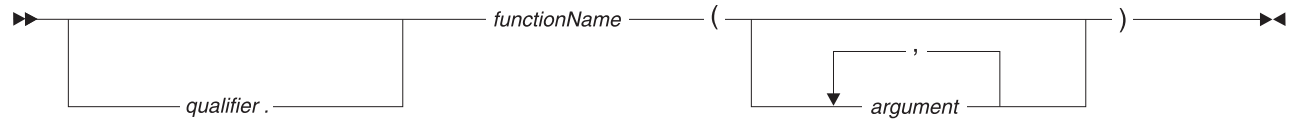
"displayMsgNum()" on page 917

"Validation properties" on page 68

---

## Function invocations

A function invocation runs an EGL-generated function or a system function. When the invoked function ends, processing continues either with the statement that follows the invocation or (in complex cases) with the next process required in an expression or in a list of arguments.



### *qualifier*

One of the following symbols:

- The name of the library in which the function resides; or
- The name of the package in which the function resides, optionally followed by a period and the name of the library in which the function resides.
- *this* (identifies a function in the current program)

For details on the circumstances in which the qualifier is unnecessary, see *References to parts*.

### *function name*

Name of the invoked function.

### *argument*

One of the following:

- Literal
- Constant
- Variable
- A more complex numeric, text, or datetime expression, potentially including a function invocation or substring; however, the access modifier for a non-reference parameter must be IN

You can pass a reference variable or an expression that evaluates to a reference, as noted in *Reference variables and NIL in EGL*.

The effect on a non-reference variable that is passed as an argument to an EGL-generated function depends on whether the corresponding parameter is modified with IN, OUT, or INOUT. For details, see *Function parameters*.

If the invoked function returns a value, you can use the invocation in these ways:

- As a complete EGL statement (in which case the function does not return a value and is followed by a semicolon).
- As the source value in an assignment statement.
- As an operand in an expression.
- As an argument in the invocation of a function

A function invoked as in a function invocation can cause the side effect of a variable changing values when the same variable is used in the function or even in the function invocation itself. Consider this example, which assumes that the function Sum is returning the sum of three arguments and that the function Increment is adding one to a passed argument:

```
b INT = 1;  
x INT = Sum( Increment(b), b, Increment(b) );
```



If the argument to Increment is related to a parameter modified with INOUT, the effect of the preceding statements is as follows:

- `b = 1`
- The first (leftmost) invocation of Increment revises the value of `b`, which equals 2 on the return from Increment
- The second argument in the invocation of Sum is 2
- The second (rightmost) invocation of Increment revises the value of `b`, which equals 3 on the return from Increment
- After Sum runs, `x` receives the value 7 because the logic in that function used the values 2, 2, and 3

If the second argument in the invocation of Sum is related to a parameter modified with INOUT, evaluation of that argument occurs after both invocations of Increment. The effect of the preceding code is as follows:

- `b = 1`
- The first (leftmost) invocation of Increment revises the value of `b`, which equals 2 on the return from Increment
- The second (rightmost) invocation of Increment revises the value of `b`, which equals 3 on the return from Increment
- The logic in Sum begins to run, and only then is the memory associated with the second argument referenced; the value in that memory is equal to 3
- After Sum runs, `x` receives the value 8 because the logic in that function used the values 2, 3, and 3

The general rule is that side effects can be identified by reference to the usual order of evaluation of expressions, which is left to right but can be overridden by parentheses. The use of INOUT is a further complication, as shown.

When the access modifier of a parameter is IN or OUT, the compatibility rules are as described in *Assignment compatibility*. When the access modifier of a parameter is INOUT or when the parameter is in the `onPageLoad` function of a `pageHandler`, the compatibility rules are as described in *Reference compatibility*.

Other rules also apply:

#### **literals**

If the access modifier is IN or INOUT, you can code a literal as the argument. The EGL-generated code creates a temporary variable of the parameter type, initializes that variable with the value, and passes the variable to the function.

#### **fixed record**

If the argument is a fixed record, the parameter must be a fixed record.

The following rules apply to fixed records that are not of type `basicRecord`:

- The type of the argument and parameter must be identical
- The access modifier must be of type INOUT

In relation to fixed records that are of type `basicRecord`, the type of the argument and parameter can vary:

- If the access modifier is of type IN, the size of the argument must be greater than or equal to the size of the parameter.
- If the access modifier is of type OUT or INOUT, the size of the argument must be less than or equal to the size of the parameter.

### Related concepts

"Function part" on page 150

"References to parts" on page 23

"Syntax diagram for EGL statements and commands" on page 884

### Related tasks

"Assignments" on page 456

### Related reference

"Assignment compatibility in EGL" on page 451

"EGL statements" on page 88

"Function parameters" on page 616

"Function part in EGL source format" on page 621

"Primitive types" on page 34

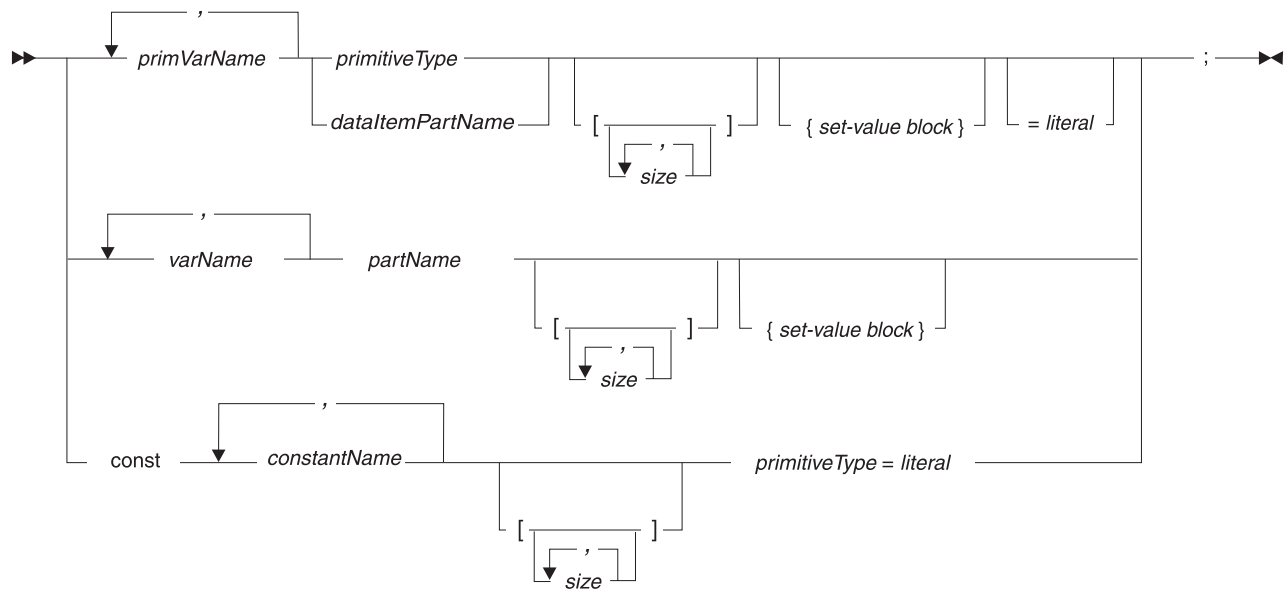
"Reference compatibility in EGL" on page 862

"Reference variables and NIL in EGL" on page 863

---

## Function variables

The syntax diagram for each variable in a function is as follows:



#### *primVarName*

Specifies the name of a local primitive variable. For details on usage in the function, see *References to variables and constants*. For other rules, see *Naming conventions*.

#### *primitiveType*

The type of a primitive field. Depending on the type, the following information may be required:

- The parameter's length, which is an integer that represents the number of characters or digits in the memory area.
- For some numeric types, you may specify an integer that represents the number of places after the decimal point. The decimal point is not stored with the data.

- For an item of type INTERVAL or TIMESTAMP, you may specify a datetime mask, which assigns a meaning (such as "year digit") to a given position in the item value.

#### *dataItemPartName*

The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

#### *size*

Number of elements in the array. If you specify the number of elements, the array is initialized with that number of elements.

#### *set-value block*

For details, see *Overview of EGL properties* and *Set-value blocks*

#### *= literal*

Specifies the initial value of the primitive variable.

#### *varName*

Name of the variable, which can be of any type that is based on a part.

#### *partName*

Name of a part that is visible to the program or is predefined. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

#### **const** *constantName primitiveType=literal*

Name, type, and value of a constant. Specify a quoted string (for a character type); a number (for a numeric type); or an array of appropriately typed values (for an array). Examples are as follows:

```
const myString String = "Great software!";
const myArray BIN[] = [36, 49, 64];
const myArray02 BIN[] [] = [[1,2,3],[5,6,7]];
```

For the rules of naming, see *Naming conventions*.

#### **Related concepts**

"Function part" on page 150

"Parts" on page 19

"References to parts" on page 23

"References to variables in EGL" on page 59

"Syntax diagram for EGL statements and commands" on page 884

"Typedef" on page 28

#### **Related tasks**

"Function part in EGL source format" on page 621

#### **Related reference**

"Arrays" on page 75

"INTERVAL" on page 42

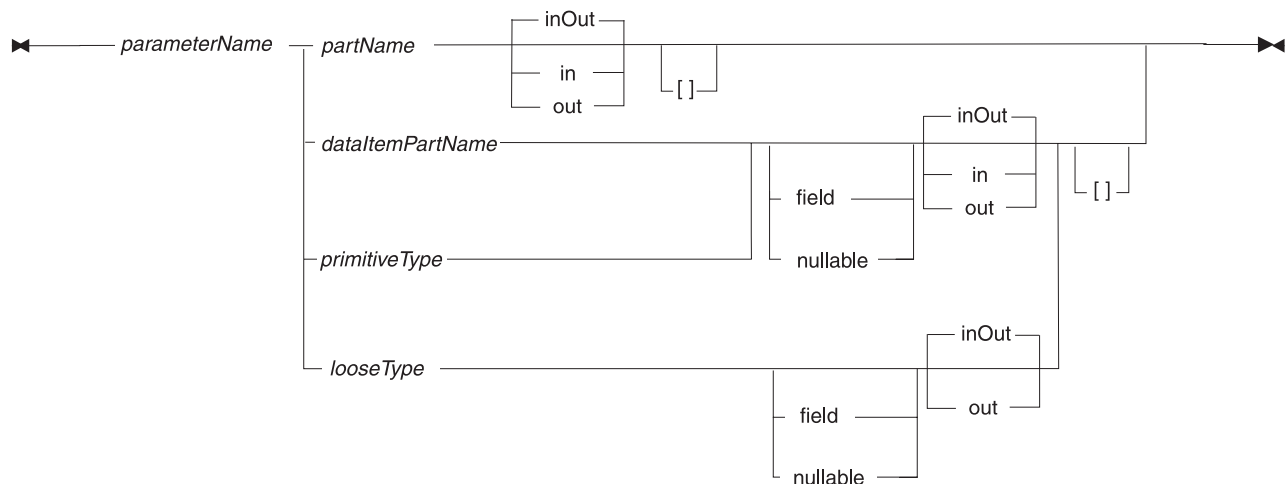
"Naming conventions" on page 778

"TIMESTAMP" on page 44

---

## Function parameters

The syntax diagram for a function parameter is as follows:



#### *parameterName*

Specifies the name of a parameter. For rules, see *Naming conventions*.

If you specify the modifier **inOut** or **out** (as is possible for a parameter that is not based on a reference type), any changes that are made to the parameter value are available in the invoking function. Those modifiers are described later and in the section “Implications of inOut and the related modifiers” on page 620. For details on how reference types are handled, see *Reference variables and NIL in EGL*.

A parameter can be passed as an argument to another function. A parameter is not otherwise visible to functions invoked by the function containing the parameter.

A parameter that ends with brackets ([ ]) is a dynamic array, and the other specifications declare aspects of each element of that array.

#### *partName*

Name of a part, which may be a record part, dictionary, arrayDictionary, or a reference type.

The following statements apply to input or output (I/O) against a fixed record:

- A fixed record passed from another function in the same program includes record state such as the I/O error value *endOfFile*, but only if the record is of the same record type as the parameter. Similarly, any change in the record state is returned to the invoker, so if you perform I/O against a record parameter, any tests on that record can occur in the current function, in the invoker, or in a function that is called by the current function.

Library functions do not receive record state.

- Any I/O operation performed against the fixed record uses the record properties specified for the parameter, not the record properties specified for the argument.
- For fixed records of type *indexedRecord*, *mqRecord*, *relativeRecord*, or *serialRecord*, the file or message queue associated with the record declaration is treated as a run-unit resource rather than a program resource. Local record declarations share the same file (or queue) whenever the record property **fileName** (or **queueName**) has the same value. Only one physical file at a time can be associated with a file or queue name no matter how many records are associated with the file or queue in the run unit, and EGL enforces this rule by closing and reopening files as appropriate.

### **inOut (not supported for reference types)**

The function receives the argument value as an input, and the invoker receives any changes to the parameter when the function ends. If the argument is a literal or constant, however, the argument is treated as if the modifier **in** were in effect.

The **inOut** modifier is necessary if the parameter is a field and you specify the modifier **field**, which indicates that the parameter has testable, form-field attributes such as *blanks* or *numeric*.

If the parameter is a record, the following rules apply:

- If you intend to use that record to access a file or database in the current function (or in a function invoked by the current function), you must specify the **inOut** modifier or accept that modifier by default
- If the type of record is the same for argument and parameter (for example, if both are serial records), the record-specific state information such as end-of-file status is available in the function and is returned to the invoker, but only if the **inOut** modifier is in effect

When you specify a limited-length string as a function parameter whose modifier is **out**, the length limit must be the same in argument and parameter.

If the **inOut** modifier is in effect, the related argument must be reference-compatible with the parameter, as described in *Reference Compatibility in EGL*.

### **in (not supported for reference types)**

The function receives the argument value as an input, but the invoker is not affected by changes made to the parameter.

You cannot use the **in** modifier for a field that has the modifier **field**. Also, you cannot specify the **in** modifier for a record that is used to access a file or database either in the current function or in a function invoked by the current function.

When you specify a limited-length string as a function parameter whose modifier is **in**, any text input is valid:

- If more characters are in the source than are valid in the target, EGL runtime truncates the copied content to fit the available length.
- If fewer characters are in the source than are valid in the target, EGL runtime pads the copied content with blanks, to the specified length.

### **out (not supported for reference types)**

The function does not receive the argument value as an input; rather, the input value is initialized according to the rules described in *Data Initialization*. The value of the parameter is assigned to the argument when the function returns.

If the argument is a literal or constant, the argument is treated as if the modifier **in** were in effect.

You cannot use the **out** modifier for a parameter that has the modifier **field**. Also, you cannot specify the **out** modifier for a record that is used to access a file or database either in the current function or in a function invoked by the current function.

When you specify a limited-length string as a function parameter whose modifier is **out**, the length limit must be the same in argument and parameter.

#### *dataItemPartName*

A DataItem part that is visible to the function and that is acting as a typedef (a model of format) for a parameter.

#### *primitiveType*

The type of a primitive field. Depending on the type, the following information may be required:

- The parameter's length, which is an integer that represents the number of characters or digits in the memory area.
- For some numeric types, you may specify an integer that represents the number of places after the decimal point. The decimal point is not stored with the data.
- For a field of type INTERVAL or TIMESTAMP, you may specify a datetime mask, which assigns a meaning (such as "year digit") to a given position in the item value.

In a service, a parameter cannot be of type ANY, BLOB, or CLOB.

#### *looseType*

A loose type is a special kind of primitive type that is used only for function parameters. You use this type if you wish the parameter to accept a range of argument lengths. The benefit is that you can invoke the function repeatedly and can pass an argument of a different length each time.

Valid values are as follows:

- CHAR
- DBCHAR
- HEX
- MBCHAR
- NUMBER
- UNICODE

If you wish the parameter to accept a number of any primitive type and length, specify NUMBER as the loose type. In this case, the number passed to the parameter must not have any decimal places.

If you wish the parameter to accept a string of a particular primitive type but any length, specify CHAR, DBCHAR, MBCHAR, HEX, or UNICODE as the loose type and make sure that the argument is of the corresponding primitive type.

The definition of the argument determines what occurs when a statement in the function operates on a parameter of a loose type.

Loose types are not available in functions that are declared in *libraries* or *services*.

For details on primitive types, see *Primitive types*.

#### **field**

Indicates that the parameter has form-field attributes such as *blanks* or *numeric*. Those attributes can be tested in a logical expression.

The **field** modifier is available only if you specify the **inOut** modifier or accept the **inOut** modifier by default.

The **field** modifier is not available for function parameters in a service or in a library of type nativeLibrary.

### **nullable**

The **nullable** modifier indicates that the parameter can be set to null and that the function has access to the state information necessary to test for null in a logical expression.

The **nullable** modifier is meaningful only in the following case:

- The build descriptor option **itemsNullable** is set to *yes*; or
- Regardless of the value of that build descriptor option, the argument is a field in a non-fixed record or is a structure field in an SQL record, and the field-level property **isNullable** is set to *yes*.

If the function is in a service part or in an interface part of type basicInterface, only the second case applies because the build descriptor option **itemsNullable** is always set to **no**.

You can specify **nullable** regardless of whether the modifier **inOut**, **in**, or **out** is in effect. However, when **inOut** is in use, the following rules apply:

- An argument that is nullable is compatible with a nullable or non-nullable parameter
- An argument that is not nullable requires a non-nullable parameter because if the parameter were nullable, the function could null the parameter but not return the null indicator to the invoker

## **Implications of inOut and the related modifiers**

To better understand the modifiers **inOut**, **out**, and **in**, review the following example, which shows (in comments) the values of different variables at different points of execution.

```
program inoutpgm
  a int;
  b int;
  c int;

  function main()
    a = 1;
    b = 1;
    c = 1;

    func1(a,b,c);

    // a = 1
    // b = 3
    // c = 3
  end

  function func1(x int in, y int out, z int inout)
    // a = 1          x = 1
    // b = 1          y = 0
    // c = 1          z = 1

    x = 2;
    y = 2;
    z = 2;

    // a = 1          x = 2
    // b = 1          y = 2
    // c = 2          z = 2
```

```

func2();
func3(x, y, z);
// a = 1          x = 2
// b = 1          y = 3
// c = 3          z = 3

end

function func2()
// a = 1
// b = 1
// c = 2

end

function func3(q int in, r int out, s int inout)
// a = 1          x = unresolved   q = 2
// b = 1          y = unresolved   r = 2
// c = 2          z = unresolved   s = 2

q = 3;
r = 3;
s = 3;

// a = 1          x = unresolved   q = 3
// b = 1          y = unresolved   r = 3
// c = 3          z = unresolved   s = 3

end

```

### Related concepts

["Function part" on page 150](#)  
["Library part of type basicLibrary" on page 169](#)  
["Library part of type basicLibrary" on page 169](#)  
["Parts" on page 19](#)  
["References to parts" on page 23](#)  
["References to variables in EGL" on page 59](#)  
["Typedef" on page 28](#)

### Related reference

["Basic record part in EGL source format" on page 461](#)  
["Data initialization" on page 564](#)  
["EGL source format" on page 586](#)  
["Function part in EGL source format"](#)  
["Indexed record part in EGL source format" on page 632](#)  
["INTERVAL" on page 42](#)  
["Logical expressions" on page 593](#)  
["MQ record part in EGL source format" on page 769](#)  
["Naming conventions" on page 778](#)  
["Primitive types" on page 34](#)  
["Reference compatibility in EGL" on page 862](#)  
["Reference variables and NIL in EGL" on page 863](#)  
["Relative record part in EGL source format" on page 865](#)  
["Serial record part in EGL source format" on page 868](#)  
["SQL record part in EGL source format" on page 877](#)  
["TIMESTAMP" on page 44](#)

---

## Function part in EGL source format

You can declare functions in an EGL source file, as described in *EGL source format*.



The following example shows a program part with two embedded functions, along with a standalone function and a standalone record part:

```

Program myProgram(employeeNum INT)
{includeReferencedFunctions = yes}

// program-global variable
employees record_ws;
employeeName char(20);

// a required embedded function
Function main()

// initialize employee names
recd_init();

// get the correct employee name
// based on the employeeNum passed
employeeName = getEmployeeName(employeeNum);
end

// another embedded function
Function recd_init()
employees.name[1] = "Employee 1";
employees.name[2] = "Employee 2";
end
end

// standalone function
Function getEmployeeName(employeeNum INT) returns (CHAR(20))

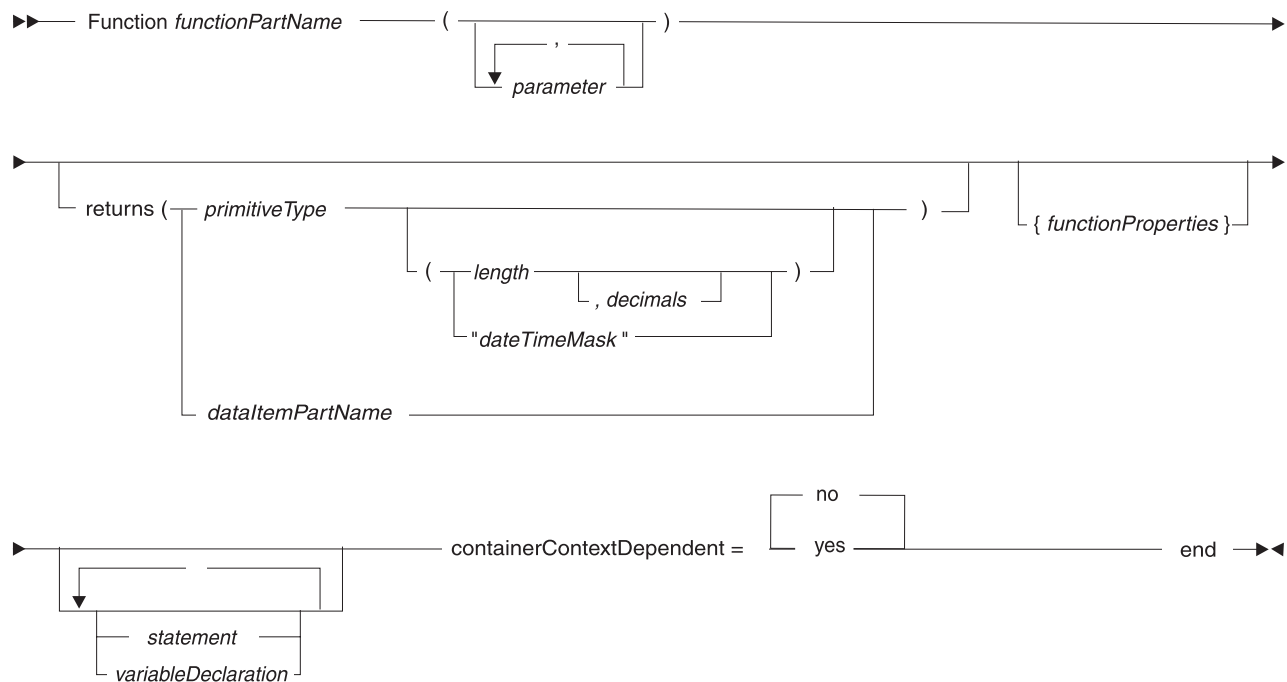
// local variable
index BIN(4);
index = syslib.size(employees.name);
if (employeeNum > index)
return("Error");
else
return(employees.name[employeeNum]);
end

end

// record part that acts as a typeDef for employees
Record record_ws type basicRecord
10 name CHAR(20)[2];
end

```

The syntax diagram for a function part is as follows:



### Function *functionPartName* ... end

Identifies the part as a function and specifies the part name. For the rules of naming, see *Naming conventions*.

#### *parameter*

A parameter, which is an area of memory that is available throughout the function and that may receive a value from the invoking function. For details on the syntax used to declare a parameter, see *Function parameters*.

#### **returns** (*returnType*)

Describes the data that the function returns to the invoker. The characteristics of the return type must match the characteristics of the variable that receives the value in the invoking function. The return type must be a primitive type.

In a service, the return type cannot be of type ANY, BLOB, or CLOB.

#### { *functionProperties* }

The properties and their types are as follows, and each is optional:

##### **@WSDL**

Is valid only if the function is in a Service part; and is meaningful only a variable that is based on that Service part is bound to a Web service.

Allows EGL to extract data from the Web Service Description Language (WSDL) definition for use in interacting with the Java JAX-RPC runtime code. The property fields and types are as follows:

##### **elementName** STRING

If this property field is present, the value becomes the name in the WSDL operation element for the function. If the property field is not present, the function name is used. The data is case sensitive: for example, the name *myFunction* is different from *MYFUNCTION*.

##### **nameSpace** STRING

This property field is ignored in the context of a function.

**isLastParamReturnValue BooleanKind**

This property field is ignored in the context of a function.

**alias STRING**

Is valid only if the function is in a library of type `nativeLibrary`. In that context, the value is the name of the DLL-based function and defaults to the EGL function name. Set the **alias** property explicitly if a validation error occurs when you name the EGL function with the name of the DLL-based function.

*dataItemPartName*

A `dataItem` part that is visible to the function and that is acting as a typedef (a model of format) for the return value.

*primitiveType*

The primitive type of the data returned to the invoker.

*length*

The length of the data returned to the invoker. The length is an integer that represents the number of characters or digits in the returned value.

*decimals*

For some numeric types, you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

*"dateTimeMask"*

For `TIMESTAMP` and `INTERVAL` types, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the datetime value. The mask is present with the data at run time.

*statement*

An EGL statement, as described in *EGL statements*. Most end with a semicolon.

*variableDeclaration*

A variable declaration, as described in *Function variables*.

*containerContextDependent*

An indication of whether to extend the namespace used to resolve the functions that are invoked by the function being declared. The default is *no*.

This indicator is for use in code that was migrated from VisualAge Generator. For details, see *containerContextDependent*.

**Related concepts**

- "EGL projects, packages, and files" on page 15
- "Function part" on page 150
- "Import" on page 33
- "Library part of type `basicLibrary`" on page 169
- "Library part of type `basicLibrary`" on page 169
- "Parts" on page 19
- "References to parts" on page 23
- "References to variables in EGL" on page 59
- "Syntax diagram for EGL statements and commands" on page 884
- "Typedef" on page 28

**Related reference**

- "Arrays" on page 75
- "`containerContextDependent`" on page 557
- "EGL statements" on page 88

“Function invocations” on page 613  
 “Function parameters” on page 616  
 “Function variables” on page 615  
 “INTERVAL” on page 42  
 “I/O error values” on page 638  
 “Naming conventions” on page 778  
 “Primitive types” on page 34  
 “TIMESTAMP” on page 44

## Generated output

The next table lists the generated output. For details on the names given to each kind of output file, see *Generated output (reference)*.

Output type	Purpose	Generation type
Build plan	Lists the code-preparation steps that will occur on the target platform	Java or Java wrapper
Enterprise JavaBean (EJB) session bean	Runs in an EJB container	Java wrapper
Java program and related classes	Runs either outside of J2EE or in the context of a J2EE client application, web application, or EJB container	Java
Java wrapper	Invokes an EGL-generated program from non-EGL-generated Java code	Java wrapper
J2EE environment file	Provides entries for insertion into the Java deployment descriptor	Java
Library (generated output)	Provides functions and values for use by other generated output	Java
Linkage properties file	Guides how calls are made from generated Java code, but only if decisions are final at deployment time rather than generation time	Java or Java wrapper
PageHandler part	Creates output that controls a user’s runtime interaction with a Web page	Java
Program properties file	Contains Java runtime properties in a format that is accessible only when you are debugging a Java program in a non-J2EE Java project	Java
Results file	Gives status information on the code-preparation steps that occurred on the target platform	Java or Java wrapper

### Related concepts

“Introduction to EGL” on page 1  
 “Java program, PageHandler, and library” on page 414  
 “Java runtime properties” on page 431  
 “Runtime configurations” on page 11

### Related tasks

“Building EGL output” on page 413

### Related reference

“Generated output (reference)” on page 626

---

## Generated output (reference)

The output of EGL generation largely depends on whether you are generating Java, or a Java wrapper. The next table shows the file names of generated output that do not come from a specific EGL part.

Output type	File name
"Build plan" on page 413	<i>alias</i> BuildPlan.xml
"Enterprise JavaBean (EJB) session bean" on page 402	<i>alias</i> EJBHome.java for the home interface, <i>alias</i> EJB.java for the remote bean interface, and <i>alias</i> EJBBean.java for the bean implementation
"J2EE environment file" on page 440	<i>alias</i> -env.txt
"Program properties file" on page 433	<i>alias</i> .properties
"Results file" on page 414	<i>alias</i> _Results_timestamp.xml

### *alias*

The alias, if any, that is specified in the program part. If the alias is not specified, the name of the program part is used but is truncated (if necessary) to the maximum number of characters allowed in the runtime environment.

Other characteristics of *alias* are determined by the kind of output:

- If you are generating a Java program, the case of each letter in *alias* is taken without change from the source code
- If you are generating a Java wrapper, the rules for naming the wrapper and EJB session bean are as follows:
  - The first letter in *alias* is uppercase
  - Every subsequent letter is lower case, with this exception: any underscore or hyphen is eliminated, and the subsequent letter is uppercase

### *timestamp*

The date and time when the file was created. The format reflects the settings on the development operating system.

For details on file names, see the appropriate reference topic:

- 
- "Output of Java program generation" on page 781
- "Output of Java wrapper generation" on page 782

### Related concepts

"Build plan" on page 413

"Enterprise JavaBean (EJB) session bean" on page 402

"Generated output" on page 625

"Generation" on page 409

"J2EE environment file" on page 440

"Program properties file" on page 433

"Results file" on page 414

### Related reference

"Output of Java program generation" on page 781

"Output of Java wrapper generation" on page 782

---

## Generation Results view

The Generation Results view shows you code-preparation messages that are the result of generation performed in the workbench. These messages may be errors, warnings, or informational messages. This view is available only when generating from the Workbench. The format is as follows:

*msgid message*

**msgid**

Is the message identifier. For example, IWN.VAL.4610.e is the message ID for Enterprise Developer validation error number 4610.

**message**

Is the text of the message.

Generation results are displayed in the view by generatable part (program, PageHandler, form group, data table, library), with a different tab for each part. The results can be a combination of validation results and generation results.

You can open this view at any time, but it displays data only after you generate output.

If you want to remove tabs from the Generation Results view, click the drop-down arrow at the top right corner of the view, and then click one of the following options:

- **Remove Tab**
- **Remove All Tabs**
- **Remove Tabs Without Errors**

**Related concepts**

"Development process" on page 9

"Generated output" on page 625

"Generation" on page 409

**Related reference**

"Generated output (reference)" on page 626

---

## IMS-related considerations for EGL

Consider the following issues when developing EGL programs for IMS:

- "Transfers to and from EGL-generated IMS MPPs"
- "Transfers to and from IMSADF programs" on page 628

### Transfers to and from EGL-generated IMS MPPs

*transfers to and from EGL-generated IMS MPPs* In IMS/VS, one message processing program (MPP) can invoke another in either of two ways:

**Immediate message switch**

In an immediate message switch, a program passes control directly to another transaction without first responding to the originating terminal. Even if the transferring program is conversational, the IMS scratch pad area (SPA) is not used during transfer; instead, the program submits data to an alternate PSB whose destination is set to the receiving transaction.

If the transferring program is generated by EGL, the transfer is accomplished by one of the following language elements:

- The **transfer** statement of type *transfer to transaction*; or

- The system function **SysLib.startTransaction**, which can start only a non-conversational program.

#### Deferred message switch

In a deferred message switch, a program displays a text form on the terminal so that, when the user submits the form, IMS starts another transaction. The mechanism of transfer depends on whether the transferring program is conversational or nonconversational:

- A transferring, conversational program modifies the SPA to include the transaction name of the receiving program.
- A transferring, nonconversational program includes the new transaction name on the text form so that the name is in the first eight bytes of the message received by IMS when the user submits the form.

In either case, the modified data tag in the text form should be set for all input fields on the form. Otherwise, the following consequences apply:

- If the user submits the form without changing the value in a given field, the data in that field is not processed by the receiving program
- If the user who is reviewing the form requests presentation of a help form and then re-displays the original form, the form-field values displayed on the original form are from the defaults in that form rather than from the sending program

If the transferring program is generated by EGL, the transfer is accomplished by the **show** statement that includes a **returning** clause.

The use of the property **inputForm** in a receiving, EGL-generated MPP determines which type of message switch is valid:

- If **inputForm** is not specified, an immediate message switch is required.
- If **inputForm** is specified, either type of message switch is possible, but use of a deferred switch is more efficient. If the immediate switch is used, the form identified in **inputForm** is displayed automatically by the transferred-to program, but the receiving program is processed twice — once to display the form and once to read the user data.

When you transfer control from an EGL-generated IMS/VS program to an EGL-generated IMS/VS program, the conversational status of the two programs must be the same: either both must be conversational, or both must be non-conversational. To specify that an EGL-generated program has reserved a SPA, assign a positive value in the build descriptor option **spaSize**.

#### Related concepts

## Transfers to and from IMSADF programs

*transfers to and from EGL-generated IMS MPPs* In IMS/VS, one message processing program (MPP) can invoke another in either of two ways:

#### Immediate message switch

In an immediate message switch, a program passes control directly to another transaction without first responding to the originating terminal. Even if the transferring program is conversational, the IMS scratch pad area (SPA) is not used during transfer; instead, the program submits data to an alternate PSB whose destination is set to the receiving transaction.

If the transferring program is generated by EGL, the transfer is accomplished by one of the following language elements:

- The **transfer** statement of type *transfer to transaction*; or
- The system function **SysLib.startTransaction**, which can start only a non-conversational program.

#### Deferred message switch

In a deferred message switch, a program displays a text form on the terminal so that, when the user submits the form, IMS starts another transaction. The mechanism of transfer depends on whether the transferring program is conversational or nonconversational:

- A transferring, conversational program modifies the SPA to include the transaction name of the receiving program.
- A transferring, nonconversational program includes the new transaction name on the text form so that the name is in the first eight bytes of the message received by IMS when the user submits the form.

In either case, the modified data tag in the text form should be set for all input fields on the form. Otherwise, the following consequences apply:

- If the user submits the form without changing the value in a given field, the data in that field is not processed by the receiving program
- If the user who is reviewing the form requests presentation of a help form and then re-displays the original form, the form-field values displayed on the original form are from the defaults in that form rather than from the sending program

If the transferring program is generated by EGL, the transfer is accomplished by the **show** statement that includes a **returning** clause.

The use of the property **inputForm** in a receiving, EGL-generated MPP determines which type of message switch is valid:

- If **inputForm** is not specified, an immediate message switch is required.
- If **inputForm** is specified, either type of message switch is possible, but use of a deferred switch is more efficient. If the immediate switch is used, the form identified in **inputForm** is displayed automatically by the transferred-to program, but the receiving program is processed twice — once to display the form and once to read the user data.

When you transfer control from an EGL-generated IMS/VS program to an EGL-generated IMS/VS program, the conversational status of the two programs must be the same: either both must be conversational, or both must be non-conversational. To specify that an EGL-generated program has reserved a SPA, assign a positive value in the build descriptor option **spaSize**.

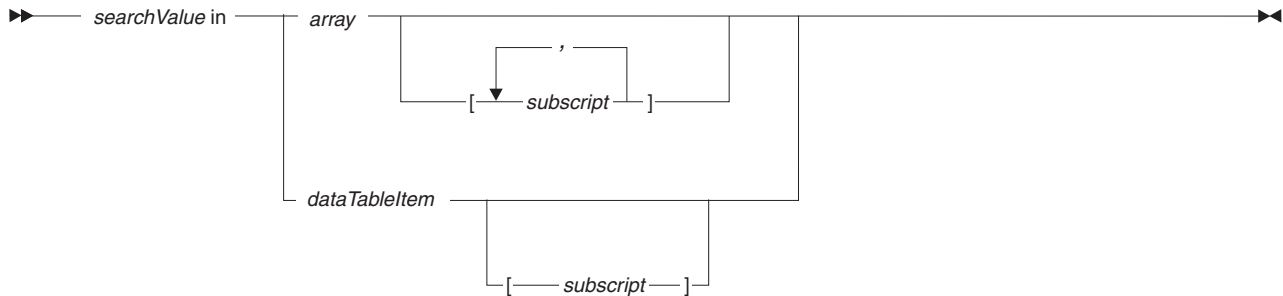
#### Related concepts

---

## in operator

The operator **in** is a binary operator used in an elementary logical expression that has the following format:





#### *searchValue*

A literal or item, but not a system variable.

**array** A one-dimensional or multidimensional array. The operator **in** operates on a one-dimensional array, which may be an element of a multidimensional array.

#### **subscript**

An integer, or an item (or system variable) that resolves to an integer. The value of a subscript is an index that refers to a specific element in an array.

An item used as a subscript of an array can not itself be an array element. In each of the following examples, `myItemB[1]` is both a subscript and an array element; as a result, the following syntax is *not* valid:

```
/* the next syntax is not valid */
myItemA[myItemB[1]]

// this next syntax is not valid; but only
// because myItemB is myItemB[1], the
// first element of a one-dimensional array
myItemA[myItemB]
```

#### **dataTableItem**

The name of a `dataTable` item. The item represents a column in the data table. The **in** operator interacts with that column as if the column were a one-dimensional array.

The logical expression resolves to true if the generated program finds the search value. The search begins at the element identified by the last array subscript. If *array* is a one-dimensional array, the last subscript is optional and defaults to 1. If *array* is a multidimensional array, the following statements are true:

- A subscript must be present for each dimension
- The generated program searches the one-dimensional array that is identified by the sequence of subscripts other than the last subscript
- The search begins at the element identified by the last subscript

In relation to both one-dimensional and multidimensional arrays, the search ends at the last element of the one-dimensional array under review.

The logical expression that includes **in** resolves to false in either of these cases:

- The search value is not found
- The value of the last subscript is greater than the number of entries in the one-dimensional array being searched

If the elementary logical expression resolves to true, the operation **in** sets the system variable **sysVar.arrayIndex** to the subscript value of the element that contains the search value. If the expression resolves to false, the operation sets **sysVar.arrayIndex** to zero.

## Examples with a one-dimensional array

Let's assume that the structure item myString is substructured to an array of three characters:

```
structureItem name="myString" length=3
structureItem name="myArray" occurs=3 length=1
```

The next table shows the effect of the operator **in** if myString is "ABC".

Logical expression	Value of expression	Value of sysVar. ArrayIndex	Comment
"A" in myArray	true	1	The subscript of a single-dimension array defaults to 1
"C" in myArray[2]	true	3	Search begins at second element
"A" in myArray[2]	false	0	The search ends at the last element

## Examples with a multidimension array

Let's assume that the array myArray01D is substructured to an array of three characters:

```
structureItem name="myArray01D" occurs=3 length=3
structureItem name="myArray02D" occurs=3 length=1
```

In this example, myArray01D is a one-dimensional array, with each element containing a string that is substructured to an array of three characters. myArray02D is a two-dimensional array, with each element (such as myArray02D[1,1]) containing a single character.

If the content of myArray01D is "ABC", "DEF", and "GHI", the content of myArray02D is as follows:

```
"A"  "B"  "C"
"D"  "E"  "F"
"G"  "H"  "I"
```

The next table shows the effect of the operator **in**.

Logical expression	Value of expression	Value of sysVar. ArrayIndex	Comment
"DEF" in myArray01D	true	2	A reference to a one-dimensional array does not require a subscript; by default, the search begins at the first element

Logical expression	Value of expression	Value of sysVar. ArrayIndex	Comment
"C" in myArray02D[1]	—	—	The expression is invalid because a reference to a multidimensional array must include a subscript for each dimension
"I" in myArray02D[3,2]	true	3	Search begins at the third row, second element
"G" in myArray02D[3,2]	false	0	Search ends at the last element of the row being reviewed
"G" in myArray02D[2,4]	false	0	The second subscript is greater than the number of columns available to search

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

### Related reference

"Arrays" on page 75

"Logical expressions" on page 593

"Operators and precedence" on page 779

"arrayIndex" on page 1065

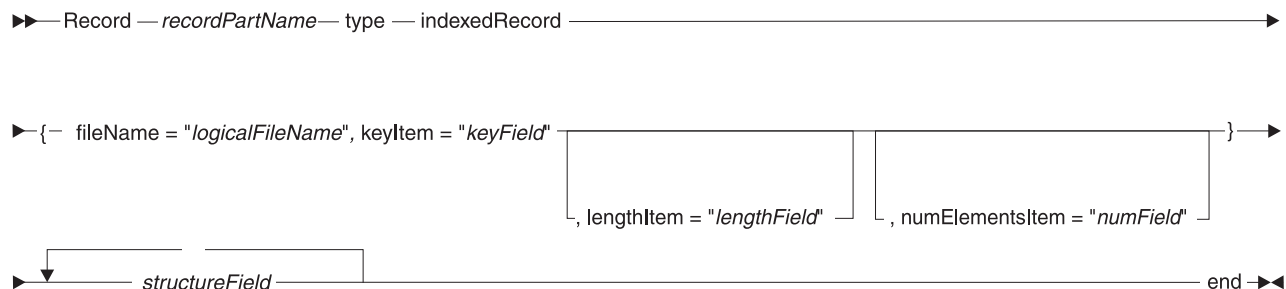
## Indexed record part in EGL source format

You declare a record part of type indexedRecord in an EGL source file, which is described in *EGL source format*.

An example of an indexed record part is as follows:

```
Record myIndexedRecordPart type indexedRecord
{
    fileName = "myFile",
    keyItem = "myKeyItem"
}
10 myKeyItem CHAR(2);
10 myContent CHAR(78);
end
```

The syntax diagram for an indexed record part is as follows:



**Record** *recordPartName* **indexedRecord**

Identifies the part as being of type `indexedRecord` and specifies the name. For rules, see *Naming conventions*.

**fileName** = *"logicalFileName"*

The file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

**keyItem** = *"keyItem"*

The key item, which can only be a structure item that is unique in the same record. You must use an unqualified reference for *keyItem*; for example, use *myItem* rather than *myRecord.myItem*. (In a function, however, you can reference that structure item as you would reference any structure item.)

**lengthItem** = *"lengthItem"*

The length item, as described in *Properties that support variable-length records*.

**numElementsItem** = *"numElementsItem"*

The number of elements item, as described in *Properties that support variable-length records*.

*structureItem*

A structure item, as described in *Structure item in EGL source format*.

**Related concepts**

"EGL projects, packages, and files" on page 15

"References to parts" on page 23

"Parts" on page 19

"Record parts" on page 135

"References to variables in EGL" on page 59

"Resource associations and file types" on page 393

"Typedef" on page 28

**Related tasks**

"Syntax diagram for EGL statements and commands" on page 884

**Related reference**

"Arrays" on page 75

"DataItem part in EGL source format" on page 566

"EGL source format" on page 586

"Function part in EGL source format" on page 621

"MQ record part in EGL source format" on page 769

"Naming conventions" on page 778

"Primitive types" on page 34

"Program part in EGL source format" on page 841

"Properties that support variable-length records" on page 860

"Relative record part in EGL source format" on page 865

"Serial record part in EGL source format" on page 868

"SQL record part in EGL source format" on page 877

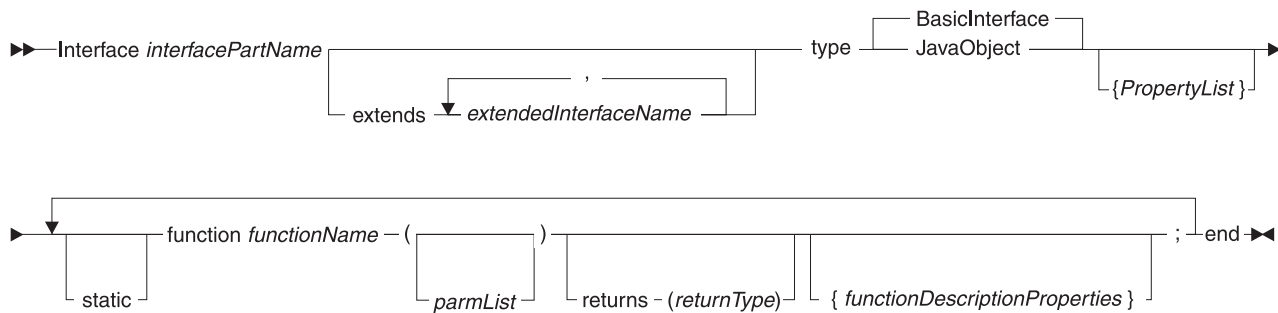
"Structure field in EGL source format" on page 880

---

## Interface part in EGL source format

For an overview of interfaces, see *EGL interfaces*. You can declare interfaces in an EGL source file, which is described in *EGL source format*.

The syntax diagram for an Interface part is here:



### Interface *interfacePartName* ... end

Identifies the part as an interface and specifies the part name. For the rules of naming, see *Naming conventions*.

### extends *extendedInterfaceName*

Indicates that the interface inherits the functions of each of the other named interfaces and of any interfaces that the named interfaces were extended from, to any level of inheritance.

If a function description in the interface has the same name as an inherited function description, the function description in the interface overrides the inherited description.

An interface can inherit more than one function having the same argument list; but a compile-time error occurs if two such functions have different return types or if one has a return type and the other does not.

An interface can extend interfaces only of the same subtype; an interface of type *JavaObject*, for example, cannot extend an interface of type *BasicInterface*.

### subtype

One of these subtypes:

#### **BasicInterface (the default)**

For accessing services

#### **JavaObject**

For accessing Java code

### *PropertyList*

List of properties. For details, see the topic of interest:

- *Interfaces of type BasicInterface*
- *Interfaces of type JavaObject*

### static

Indicates that the function is not specific to a variable of type *Interface* but can be invoked as follows:

*interfaceName.functionName*

*interfaceName*

Name of the Interface part.

*functionName*

Name of the function.

### *functionName*

Name of the function as used in EGL code.

### *parmList*

Each parameter, type, and modifier (IN, OUT, or INOUT), with one entry separated from the next by a comma.

If you are creating an interface of type `JavaObject`, a parameter can be of type `BOOLEAN`, and the built-in EGL interface `BooleanLib` allows you to convert between `Boolean` and integer values. For details, see *Interfaces of type JavaObject*.

When an EGL wizard creates an interface of type `BasicInterface` to access a Web service that returns an array or record (as is possible when the service is written in a language other than EGL), no return value is specified; instead, the wizard acts as follows:

- Adds a parameter to the end of the parameter list, specifying the type of the return value and including the OUT modifier; and
- Sets the **@WSDL** property, field **isLastParamReturnValue** to *yes*.

### **returns** (*returnType*)

Describes the data (if any) that the function returns to the invoker.

In an interface of type `JavaObject`, the return type can be `BOOLEAN`, and the built-in EGL interface `BooleanLib` allows you to convert between `Boolean` and integer values. For details, see *Interfaces of type JavaObject*.

When an EGL wizard creates an interface of type `BasicInterface` to access a Web service that returns an array or record (as is possible when the service is written in a language other than EGL), no return value is specified; instead, the wizard acts as follows:

- Adds a parameter to the end of the parameter list, specifying the type of the return value and including the OUT modifier; and
- Sets the **@WSDL** property, field **isLastParamReturnValue** to *yes*.

### *functionDescriptionProperties*

The following properties are supported:

#### **@WSDL**

Allows EGL to extract data from the Web Service Description Language (WSDL) definition for use in interacting with the Java JAX-RPC runtime code. The **@WSDL** property fields and their types are as follows:

#### **elementName String**

If this property field is present, the value must exactly match the name in the WSDL operation element for the service that is being accessed by the interface. If the property field is not present, the name of the Interface part must exactly match the name in the WSDL operation element. In either case, the match is case sensitive: for example, the name *myFunction* is different from *MYFUNCTION*.

#### **namespace String**

This property field is ignored in the context of a function description.

#### **isLastParamReturnValue BooleanKind**

Indicates whether the last parameter in *parmList* refers to the function's return value. The default is *no*, which means that the last parameter in the function description refers to the last parameter in the function itself.

When an EGL wizard creates an interface of type `BasicInterface` to access a Web service that returns an array or record (as is possible when the service is written in a language other than EGL), no return value is specified; instead, the wizard acts as follows:

- Adds a parameter to the end of the parameter list, specifying the type of the return value and including the OUT modifier; and
- Sets the **@WSDL** property, field **isLastParamReturnValue** to *yes*.

**JavaName STRING**

Specifies the name of the Java method associated with the EGL function description. The default is the function name; but the property is useful if the Java name includes characters that are not valid in EGL. For details on validity, see *Naming conventions*.

To use an interface, you must specify the package `com.ibm.egl.jsf`.

**Related concepts**

“EGL interfaces” on page 151

**Related tasks**

“Creating an EGL Interface part” on page 150

**Related reference**

“Interfaces of type BasicInterface”

“Interfaces of type JavaObject” on page 637

## Interfaces of type BasicInterface

An interface of type BasicInterface is used to access an EGL or Web service.

In relation to an Interface that accesses a Web service, you may set the **@WSDL** property, which allows EGL to extract data from the Web Service Description Language (WSDL) definition for use in interacting with the Java JAX-RPC runtime code.

The **@WSDL** property fields and their types are as follows:

**elementName STRING**

If this property field is present, the value must exactly match the name in the WSDL portType element for the service that is being accessed by the interface. If the property field is not present, the name of the Interface part must exactly match the name in the WSDL portType element. In either case, the match is case sensitive: for example, the name *myService* is different from *MYSERVICE*.

**namespace STRING**

If this property field is present, the value must exactly match the namespace specified in the WSDL portType element for the service being accessed by the interface. If the property field is not present, the value is created by starting with the string *http://* and inverting every qualifier in the package name; for example, if the package name is `com.ibm.egl`, the value of **namespace** is as follows:

```
http://egl.ibm.com
```

The value is case sensitive: for example, the namespace *http://egl.ibm.com* is different from *http://EGL.IBM.com*.

**isLastParamReturnValue BooleanKind**

This property field is ignored in the context of an Interface part.

### Related concepts

"EGL interfaces" on page 151

"EGL services and Web services" on page 158

### Related tasks

"Creating an EGL Interface part" on page 150

### Related reference

"Best practices for services and related interfaces in EGL" on page 162

"Interface part in EGL source format" on page 633

"Interfaces of type `JavaObject`"

"@xsd" on page 798

## Interfaces of type `JavaObject`

An interface of type `JavaObject` provides access to Java code. For an overview, see *EGL interfaces*. For syntax, see *Interface part in EGL source format*.

The properties of an Interface part of type `JavaObject` are these:

### **JavaName**

The name of the Java class to which the interface provides access. The default is the name of the EGL interface.

### **PackageName**

The name of the Java package in which the class resides. If the name is not specified, no package name is used and a Java compilation error is possible.

The following pre-defined EGL interfaces are available in your code when you import the EGL package `com.ibm.egl.jsf`:

### **Object**

Provides access to the Java `Object` class. The interface is this:

```
Interface Object type JavaObject
{
    JavaName = "Object",
    PackageName = "java.lang"
}
function toString() returns (String);
End
```

### **BooleanLib**

Allows you to convert values between Boolean and integer values. The interface is this:

```
Interface BooleanLib type JavaObject
{
    JavaName = "BooleanLib",
    PackageName = "com.ibm.javart.v6"
}
static function booleanToInt(var Boolean) returns (int);
static function intToBoolean(var int)
    returns (Boolean);
End
```

Two functions are available:

- **booleanToInt** returns 1 or 0, depending on the Boolean value that is passed as an argument: 1 is returned for *yes*, 0 is returned for *no*.



- **intToBoolean** returns the Boolean value *yes* or *no*, depending on the integer value that is passed as an argument: *yes* is returned for any value other than 0, *no* is returned for 0.

As is true of any static functions, you access those functions by referencing the Interface part rather than a variable of that part type:

```
package myPkg;

// required import statement
import com.ibm.egl.jsf;

myInt Int = BooleanLib.booleanToInt(yes);
```

Other pre-defined EGL interfaces let you interact with JSF controls from a pageHandler. For an overview, see *JSF component tree*.

Finally, the next table shows the mapping of EGL and Java data formats at run time, when data is transferred between an argument and a parameter or when data is returned to EGL from the invoked method.

*Table 10. Mapping of EGL primitives to Java types, for interfaces of type JavaObject*

EGL primitive type	Java type
BIGINT	long
BOOLEAN (as is available in interface function descriptions)	boolean
CHAR	char
FLOAT	double
HEX	byte
INT	int
SMALLINT	short
SMALLFLOAT	float
STRING	String

#### Related concepts

“EGL interfaces” on page 151

“Instantiation and EGL interfaces of type JavaObject” on page 155

“JSF component tree” on page 229

#### Related tasks

“Creating an EGL Interface part” on page 150

#### Related reference

“Interface part in EGL source format” on page 633

---

## I/O error values

The next table describes the EGL error values for input/output (I/O) operations that affect databases, files, and MQSeries message queues. The values associated with hard errors are available to your code only if the system variable `VGVar.handleHardIOErrors` is set to 1, as described in *Exception handling*.

Error value	Type of error	Type of Record	Meaning of error value
deadLock	Hard	SQL	Two program instances are trying to change a record, but neither can do so without system intervention.
duplicate	Soft	Indexed or Relative	Your code tried to access a record having a key that already exists, and the attempt succeeded. For details, see <i>duplicate</i> .
endOfFile	Soft	Indexed, Relative, Serial	For details, see <i>endOfFile</i> .
ioError	Hard or Soft	Any	EGL received a non-zero return code from the I/O operation.
format	Hard	Any	The accessed file is incompatible with the record definition. For details, see <i>format</i> .
fileNotAvailable	Hard	Any	fileNotAvailable is possible for any I/O operation and could indicate, for example, that another program is using the file or that resources needed to access the file are scarce.
fileNotFound	Hard	Indexed, Message queue, Relative, Serial	A file was not found.
full	Hard	Indexed, Relative, Serial	full is set in these cases: <ul style="list-style-type: none"> <li>• An indexed or serial file is full</li> </ul>
hardIOError	Hard	Any	A hard error occurred, which is any error except endOfFile, noRecordFound, or duplicate.
noRecordFound	Soft	Any	For details, see <i>noRecordFound</i> .
unique	Hard	Indexed, Relative, or SQL	UNQ indicates <i>unique</i> : your code tried to add or replace a record having a key that already exists, and the attempt failed. For details, see <i>unique</i> .

## duplicate

For an indexed or relative record, **duplicate** is set in these cases:

- An **add** statement tries to insert a record whose key or record ID already exists in the file or in an alternate index, and the insertion succeeds.
- A **replace** statement overwrites a record successfully, and the replacement values include a key that is the same as the alternate-index key of another record.
- A **get**, **get next**, or **get previous** statement reads a record successfully (or a **set** statement of the form *set record position* runs successfully), and a second record has the same key.

The **duplicate** setting is returned only if the access method returns the information, as is true on some operating systems but not on others. The option is not available during SQL database access.

## endOfFile

**endOfFile** is set in these conditions:

- Your code issues a **get next** statement for a serial or relative record when the related file pointer is at the end of the file. The pointer is at the end when the last record in the file was accessed by a previous **get** or **get next** statement.
- Your code issues a **get next** statement for an indexed record when the related file pointer is at the end of file, as occurs in these situations:
  - The last record in the file was accessed by a previous **get** or **get next** statement; or
  - The last record in the file was accessed by a previous **set** statement of type *set record position* when either of these conditions applies:
    - The key value matched the key of the last record in the file; or
    - Every byte in the key value was set to hexadecimal FF. (If a **set** statement of type *set record position* runs with a key value set to all hexadecimal FF, the statement sets the position pointer to the end of the file.)
- Your code issues a **get previous** statement for an indexed record when the related file pointer is at the beginning of file, as occurs in these situations:
  - The first record in the file was accessed by a previous **get** or **get previous** statement;
  - Your code did not previously access the same file; or
  - A **set** statement of type *set record position* ran with a key when no keys in the file were previous to that key.
- A **get next** statement attempts to retrieve data from an empty or uninitialized file into an indexed record.  
(An empty file is one from which all records have been deleted. An uninitialized file is one that has never had any records added to it.)
- A **get previous** statement attempts to retrieve data from an empty file into an indexed record.

## format

**format** can result from any kind of I/O operation and could be set for these reasons, among others:

- **Record format**  
The file format (fixed or variable length) is different from the EGL record format.
- **Record length**  
In relation to fixed-length records, the length of a record in the file is different from the length of the EGL record. In relation to variable-length records, the length of a record in the file is larger than the length of the EGL record.
- **File type**  
The file type specified for the record does not match the file type at run time.
- **Key length**  
The key length in the file is different from the key length in the EGL indexed record.
- **Key offset**  
The key position in the file is different from the key position in the EGL indexed record.

## noRecordFound

**noRecordFound** is set in these conditions:

- For an indexed record, no record is found that matches the key specified in a **get** statement.

- For EGL-generated Java, your code issues a **get next** or **get previous** statement for an indexed record when the VSAM file is empty or uninitialized.
- For a relative record, no record is found that matches the record ID specified in a **get** statement. Alternatively, a **get next** statement tries to access a record that is beyond the end of the file.
- For a SQL record, no row is found that matches the specified SELECT statement; or a **get next** statement occurs when no selected rows are left to review.

## unique

For an indexed or relative record, **unique** is set in these cases:

- An **add** statement tries to insert a record whose key or record ID already exists in the file or in an alternate index, and the insertion fails because of the duplication.
- A **replace** statement fails to overwrite a record because the replacement values include a key that is the same as the alternate-index key of another record.

The **unique** setting is returned only if the access method returns the information, as is true on some operating systems but not on others.

During SQL database access, **unique** is set when a SQL row being added or replaced has a key that already exists in a unique index. The corresponding sqlcode is -803.

### Related reference

"add" on page 661  
 "Association elements" on page 457  
 "close" on page 669  
  
 "delete" on page 673  
 "Exception handling" on page 94  
 "execute" on page 677  
 "get" on page 687  
 "get next" on page 701  
 "get previous" on page 708  
 "Logical expressions" on page 593  
 "open" on page 722  
 "prepare" on page 736  
 "replace" on page 738

---

## isa operator

The operator **isa** is a binary operator that tests whether a given expression is of a particular type. The main purpose is to test the type of the data that is held in a field of type ANY.

The operator is used in an elementary logical expression that has the following format:

*testExpression* **isa** *typeSpecification*

*testExpression*

A numeric, text, or datetime expression, which may be composed of a single field or literal.

*typeSpecification*

A type specification, which may be any of these:

- A part name.
- A primitive-type specification such as `STRING`; however, if the primitive type can be associated with a length, the length must be specified, as in these examples:
  - `BIN(9)`
  - `CHAR(5)`
 Do not include a datetime mask.
- A type specification (as described previously) followed by paired brackets. In this case, the complete specification indicates a dynamic array of a particular type, length (where appropriate), and number of dimensions.

The logical expression resolves to true if *testExpression* matches the type identified in *typeSpecification*; and otherwise resolves to false.

#### Related reference

“Arrays” on page 75

“Logical expressions” on page 593

“Operators and precedence” on page 779

## Java runtime properties (details)

The next table describes the properties that can be included in the deployment descriptor or program properties file, as well as the source of the value generated into the J2EE environment file, if any. The Java type for each property is `java.lang.String` unless the description column says otherwise.

Runtime property	Description	Source of the generated value
<code>cso.cicsj2c.timeout</code>	<p>Specifies the number of milliseconds before a timeout occurs during a call that uses protocol CICSJ2C. The default value is 30000, which represents 30 seconds. If the value is set to 0, no timeout occurs. The value must be greater than or equal to 0.</p> <p>The Java type in this case is <code>Java.lang.Integer</code>.</p> <p>The property has no effect on calls when the code is running in WebSphere 390; for details, see <i>Setting up the J2EE server for CICSJ2C calls</i>.</p>	Build descriptor option <b>cicsj2cTimeout</b>
<code>cso.linkageOptions.LO</code>	<p>Specifies the name of a linkage properties file that guides how the generated program or wrapper calls other programs. <i>LO</i> is the name of the linkage options part used at generation. For details, see <i>Deploying a linkage properties file</i>.</p>	<i>LO</i> is from the build descriptor option <b>linkage</b> ; and the default value is the name of the linkage options part followed by the extension <i>.properties</i>

Runtime property	Description	Source of the generated value
tcpiplistener.port	<p>Specifies the number of the port on which an EGL TCP/IP listener (of class CSOTcpipListener or CSOTcpipListenerJ2EE) listens. No default exists. For details, see the topics that concern <i>Setting up the TCP/IP listener</i>.</p> <p>The Java type in this case is <code>Java.lang.Integer</code>.</p>	Not generated
tcpiplistener.trace.file	<p>Specifies the name of the file in which to record the activity of one or more EGL TCP/IP listeners (each is of class CSOTcpipListener or CSOTcpipListenerJ2EE). The default file is <code>tcpiplistener.out</code>.</p>	Not generated; tracing is only for use by IBM
tcpiplistener.trace.flag	<p>Specifies whether to trace the activity of one or more EGL TCP/IP listeners (each of class CSOTcpipListener or CSOTcpipListenerJ2EE). Select one of these:</p> <ul style="list-style-type: none"> <li>• 1 for recording the activity into the file identified in property <b>tcpiplistener.trace.file</b></li> <li>• 0 (the default value) for not recording the activity</li> </ul> <p>The Java type in this case is <code>Java.lang.Integer</code>. For details, see the topics that concern <i>Setting up the TCP/IP listener</i>.</p>	Not generated; tracing is only for use by IBM
vgj.datemask. gregorian.long.locale	<p>Contains the date mask used in either of two cases:</p> <ul style="list-style-type: none"> <li>• The Java code generated for the system variable <code>VGVar.currentFormattedGregorianCalendarDate</code> is invoked; or</li> <li>• EGL validates a page item or text-form field that has a length of 10 or more, if the item property <b>dateFormat</b> is set to <code>systemGregorianCalendarDateFormat</code>.</li> </ul> <p><i>locale</i> is the code specified in property <b>vgj.nls.code</b>. In Web applications, you may change the date-mask property in use by assigning a different value to <code>sysLib.setLocale</code>.</p>	Build descriptor value for the long Gregorian date mask; the default value is specific to the locale

Runtime property	Description	Source of the generated value
vgj.datemask. gregorian.short. <i>locale</i>	Contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property <b>dateFormat</b> is set to <i>systemGregorianCalendarFormat</i> .  <i>locale</i> is the code specified in property <b>vgj.nls.code</b> . In Web applications, you may change the date-mask property in use by assigning a different value to <code>sysLib.setLocale</code> .	Build descriptor value for the short Gregorian date mask; the default value is specific to the locale
vgj.datemask. julian.long. <i>locale</i>	Contains the date mask used in either of two cases: <ul style="list-style-type: none"> <li>The Java code generated for the system variable <code>VGVar.currentFormattedJulianDate</code> is invoked; or</li> <li>EGL validates a page item or text-form field that has a length of 10 or more, if the item property <b>dateFormat</b> is set to <i>systemJulianDateFormat</i>.</li> </ul> <i>locale</i> is the code specified in property <b>vgj.nls.code</b> . In Web applications, you may change the date-mask property in use by assigning a different value to <code>sysLib.setLocale</code> .	Build descriptor value for the long Julian date mask; the default value is specific to the locale
vgj.datemask. julian.short. <i>locale</i>	Contains the date mask used when EGL validates a page item or text-form field that has a length of less than 10, if the item property <b>dateFormat</b> is set to <i>systemJulianDateFormat</i> .  <i>locale</i> is the code specified in property <b>vgj.nls.code</b> . In Web applications, you may change the date-mask property in use by assigning a different value to <code>sysLib.setLocale</code> .	Build descriptor value for the short Julian date mask; the default value is specific to the locale
vgj.default.databaseDelimiter	Specifies the symbol used to separate one value from the next in the system functions <b>SysLib.loadTable</b> and <b>SysLib.unLoadTable</b> . The default value is a pipe ( <code> </code> ).	Build descriptor option <b>dbContentSeparator</b>
vgj.default.dateFormat	Sets the initial value of system variable <b>StrLib.defaultDateFormat</b> ; for details on valid values, see <i>Date, time, and timestamp specifiers</i>	Build descriptor option <b>defaultDateFormat</b>

Runtime property	Description	Source of the generated value
vgj.defaultI4GLNativeLibrary	Specifies the DLL name accessed by a library of type nativeLibrary. The property is required if you did not specify the library property <b>dllName</b>	
vgj.default.moneyFormat	Sets the initial value of system variable <b>StrLib.defaultMoneyFormat</b> ; for details on valid values, see <i>formatNumber()</i>	Build descriptor option <b>defaultMoneyFormat</b>
vgj.default.numericFormat	Sets the initial value of system variable <b>StrLib.defaultNumericFormat</b> ; for details on valid values, see <i>formatNumber()</i>	Build descriptor option <b>defaultNumericFormat</b>
vgj.default.timeFormat	Sets the initial value of system variable <b>StrLib.defaultTimeFormat</b> ; for details on valid values, see <i>Date, time, and timestamp specifiers</i>	Build descriptor option <b>defaultTimeFormat</b>
vgj.default.timestampFormat	Sets the initial value of system variable <b>StrLib.defaultTimestampFormat</b> ; for details on valid values, see <i>Date, time, and timestamp specifiers</i>	Build descriptor option <b>defaultTimestampFormat</b>



Runtime property	Description	Source of the generated value
vgj.jdbc.database.SN	<p>Specifies the JDBC database name that is used when a database connection is made by way of the system function sysLib.connect or VGLib.connectionService.</p> <p>The meaning of the value is different for J2EE connections as compared with standard (non-J2EE) connections:</p> <ul style="list-style-type: none"> <li>• In relation to J2EE connections (as is needed in a production environment), the value is the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB</li> <li>• In relation to a standard JDBC connection (as may be used for debugging), the value is the connection URL; for example, jdbc:db2:MyDB</li> </ul> <p>You must customize the name of the property itself when you specify a substitution value for SN, at deployment time. The substitution value in turn must match either the server name that is included in the invocation of VGLib.connectionService or the database name that is included in the invocation of sysLib.connect.</p>	Build descriptor value for the database name that you want to associate with the specified "server name"
vgj.jdbc.default.database.autoCommit	<p>Specifies whether a commit occurs after every change to the default database. Valid values are true and false, as described in <i>sqlCommitControl</i>.</p>	Build descriptor option <b>sqlCommitControl</b>

Runtime property	Description	Source of the generated value
vgj.jdbc.default.database. <i>programName</i>	<p>Specifies the default database name that is used for an SQL I/O operation if no prior database connection exists. EGL includes the program name (or program alias, if any) as a substitution value for <i>programName</i> so that each program has its own default database. The program name is optional, however, and a property named <code>vgj.jdbc.default.database</code> is used as a default for any program that is not referenced in a program-specific property of this kind.</p> <p>The meaning of the value in the property itself is different for J2EE connections as compared with non-J2EE connections:</p> <ul style="list-style-type: none"> <li>• In relation to J2EE connections, the value is the name to which the datasource is bound in the JNDI registry; for example, <code>jdbc/MyDB</code></li> <li>• In relation to a standard JDBC connection, the value is the connection URL; for example, <code>jdbc:db2:MyDB</code></li> </ul>	<p>Depends on the connection type:</p> <ul style="list-style-type: none"> <li>• For J2EE connections, build descriptor option <b>sqlJNDIName</b></li> <li>• For non-J2EE connections, build descriptor option <b>sqlDB</b></li> </ul>
vgj.jdbc.default.password	<p>Specifies the password for accessing the database connection identified in <b>vgj.jdbc.default.database</b>.</p> <p>To avoid exposing passwords in the J2EE environment file, do one of the following tasks:</p> <ul style="list-style-type: none"> <li>• Specify a password in program and function scripts by using the system function <code>sysLib.connect</code> or <code>VGLib.connectionService</code>; or</li> <li>• Include a user ID and password in the datasource specification in the web application server, as described in <i>Setting up a J2EE JDBC connection</i>.</li> </ul>	Build descriptor option <b>sqlPassword</b>
vgj.jdbc.default.userid	Specifies the userid for accessing the database connection identified in <b>vgj.jdbc.default.database</b> .	Build descriptor option <b>sqlID</b>
vgj.jdbc.drivers	Specifies the driver class for accessing the database connection identified in <b>vgj.jdbc.default.database</b> . This property is not present in the deployment descriptor or J2EE environment file and is used only for a standard (non-J2EE) JDBC connection.	Build descriptor option <b>sqlJDBCClass</b>

Runtime property	Description	Source of the generated value
vgj.messages.file	<p>Specifies a properties file that includes messages you create or customize. The file is searched in these cases:</p> <ul style="list-style-type: none"> <li>• When the EGL runtime responds to the invocation of function <code>SysLib.getMessage</code>, which returns a message that you created; for details, see <i>SysLib.getMessage</i></li> <li>• When EGL runtime is handling a consoleUI application and attempts to present help or comment text from a file identified in the system variable <b>ConsoleLib.messageResource</b>, but that variable has no value.</li> <li>• When EGL attempts to display a Java runtime message, as explained in <i>Message customization for EGL runtime messages</i></li> </ul>	Build descriptor option <b>userMessageFile</b>
vgj.nls.code	<p>Specifies the three-letter NLS code of the program. For a list of valid values, see <code>targetNLS</code>.</p> <p>If the property is not set, these rules apply:</p> <ul style="list-style-type: none"> <li>• The value defaults to the NLS code that corresponds to the default Java locale</li> <li>• The value is <code>ENU</code> if the default Java locale does not correspond to any of the NLS codes supported by EGL</li> </ul>	Build descriptor option <b>targetNLS</b>
vgj.nls.currency	Specifies the character used as a currency symbol. The default is determined by the locale associated with <b>vgj.nls.code</b> .	Build descriptor option <b>currencySymbol</b>
vgj.nls.number.decimal	Specifies the character used as a decimal symbol. The default is determined by the locale associated with <b>vgj.nls.code</b> .	Build descriptor option <b>decimalSymbol</b>

Runtime property	Description	Source of the generated value
vgj.properties.file	<p>Used only if the first program in a non-J2EE run unit was generated with VisualAge Generator or with an EGL version that preceded 6.0.</p> <p><b>vgj.properties.file</b> specifies an alternate properties file. The file is used throughout a non-J2EE run unit in place of any non-global program properties file. Use of the global file is unaffected. (In run units whose first program was generated with the older EGL or with VisualAge Generator, the global file is called <b>vgj.properties</b>.)</p> <p>The file referenced by the property <b>vgj.properties.file</b> is used only if you include that property in a command-line directive, as in this example:</p> <pre>java -Dvgj.properties.file= c:\new.properties</pre> <p>The value of <b>vgj.properties.file</b> includes the fully qualified path to the properties file.</p> <p>Specifying the property <b>vgj.properties.file</b> in a properties file has no effect.</p>	
vgj.ra.QN.conversionTable	<p>Specifies the name of the conversion table used by a generated Java program during access of the MQSeries message queue identified by <i>QN</i>. Valid values are <i>programControlled</i>, <i>NONE</i>, or a conversion table name. The default is <i>NONE</i>.</p>	Resource associations property <b>conversionTable</b>
vgj.ra.FN.fileType	<p>Specifies the type of file associated with <i>FN</i>, which is a file or queue name identified in the record part. The property value is <i>seqws</i> or <i>mq</i>, as described in <i>Record and file type cross-reference</i>.</p> <p>You must specify this deployment descriptor property for each logical file that your program uses.</p>	Resource associations property <b>fileType</b>

Runtime property	Description	Source of the generated value
vgj.ra.FN.replace	<p>Specifies the effect of an add statement on a record associated with <i>FN</i>, which is a file name identified in a record. Select one of two values:</p> <ul style="list-style-type: none"> <li>• 1 if the statement replaces the file record</li> <li>• 0 (the default) if the statement appends a record to the file</li> </ul> <p>The Java type in this case is <code>java.lang.Integer</code>.</p>	Resource associations property <b>replace</b>
vgj.ra.FN.systemName	<p>Specifies the name of the physical file or message queue associated with <i>FN</i>, which is a file or queue name identified in the record part.</p> <p>You must specify this deployment descriptor property for each logical file that your program uses.</p>	Resource associations property <b>systemName</b>
vgj.ra.FN.text	<p>Specifies whether to cause a generated Java program to do the following when accessing a file by way of a serial record:</p> <ul style="list-style-type: none"> <li>• Append end-of-line characters during the <b>add</b> operation. On non-UNIX platforms, those characters are a carriage return and linefeed; on UNIX platforms, the only character is a linefeed.</li> <li>• Remove end-of-line characters during the <b>get next</b> operation.</li> </ul> <p><i>FN</i> is the file name associated with the serial record.</p> <p>Select one of these values:</p> <ul style="list-style-type: none"> <li>• 1 for make the changes</li> <li>• 0 (the default) for refrain from making the changes</li> </ul> <p>The Java type in this case is <code>java.lang.Integer</code>.</p>	Resource associations property <b>text</b>
vgj.trace.device.option	<p>Destination of trace data, if any. Select one of these values:</p> <ul style="list-style-type: none"> <li>• 0 for write to <code>System.out</code></li> <li>• 1 for write to <code>System.err</code></li> <li>• 2 (the default) for write to the file specified in <b>vgj.trace.device.spec</b> with this exception: for VSAM I/O traces, write to <code>vsam.out</code></li> </ul> <p>The Java type in this case is <code>java.lang.Integer</code>.</p>	The generated value, if any, is 2

Runtime property	Description	Source of the generated value
vgj.trace.device.spec	Specifies the name of the output file if <b>vgj.trace.device.option</b> is set to 2. An exception is that VSAM I/O traces are written to vsam.out.	The generated value, if any, is vgjtrace.out
vgj.trace.type	<p>Specifies the runtime trace setting. Sum the values of interest to get the tracing you want:</p> <ul style="list-style-type: none"> <li>• -1 for trace all</li> <li>• 0 for no trace (the default)</li> <li>• 1 for general trace, including function invocations and call statements</li> <li>• 2 for system functions that handle math</li> <li>• 4 for system functions that handle strings</li> <li>• 16 for data passed on a call statement</li> <li>• 32 for the linkage options used on a call</li> <li>• 128 for jdbc I/O</li> <li>• 256 for file I/O</li> <li>• 512 for all the properties except vgj.jdbc.default.password</li> </ul> <p>The Java type in this case is java.lang.Integer.</p>	The generated value, if any, is 0

### Related concepts

“Java runtime properties” on page 431

“Library part of type basicLibrary” on page 169

“Linkage properties file” on page 447

### Related tasks

“Deploying a linkage properties file” on page 447

“Setting up a J2EE JDBC connection” on page 445

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

“Setting up the TCP/IP listener for a called appl in a J2EE appl client module” on page 442

“Setting up the TCP/IP listener for a called non-J2EE application” on page 436

“Understanding how a standard JDBC connection is made” on page 309

### Related reference

“callLink element” on page 499

“cicsj2cTimeout” on page 470

“connect()” on page 1025

“connectionService()” on page 1047

“currentFormattedGregorianCalendar” on page 1078

“currentFormattedJulianDate” on page 1079

“currentShortGregorianCalendar” on page 1081

“currentShortJulianDate” on page 1082

“Date, time, and timestamp format specifiers” on page 46

“decimalSymbol” on page 471  
 “defaultDateFormat (EGL system variable)” on page 1004  
 “defaultMoneyFormat (EGL system variable)” on page 1004  
 “defaultNumericFormat (EGL system variable)” on page 1005  
 “defaultTimeFormat (system variable)” on page 1005  
 “defaultTimestampFormat (EGL system variable)” on page 1005  
 “formatNumber()” on page 1007  
 “getMessage()” on page 1032  
 “linkage” on page 484  
 “Linkage properties file (details)” on page 764  
 “loadTable()” on page 1033  
 “Message customization for EGL Java run time” on page 768  
 “Record and file type cross-reference” on page 860  
 “setLocale()” on page 1038  
 “sqlCommitControl” on page 490  
 “sqlDB” on page 490  
 “sqlID” on page 491  
 “sqlJDBCClass” on page 491  
 “sqlJNDIName” on page 492  
 “sqlPassword” on page 492  
 “targetNLS” on page 495  
 “unloadTable()” on page 1043

---

## Java wrapper classes

When you request that a program part be generated as a Java wrapper, EGL produces a wrapper class for each of the following:

- The generated program
- Each fixed record or form that is declared as a parameter in that program
- Each dynamic array that is declared as a parameter; and if the array is array of fixed records, the class for the dynamic array class is in addition to the class for the fixed record itself
- Each structure item that has these characteristics:
  - Is in one of the fixed records or forms for which a wrapper class is generated
  - Has at least one subordinate structure item; in other words, is substructured
  - Is an array; in this case, a substructured array

An example of a fixed record part with a substructured array is as follows:

```

Record myPart type basicRecord
  10 MyTopStructure CHAR(20) [5];
    20 MyStructureItem01 CHAR(10);
    20 MyStructureItem02 CHAR(10);
end
  
```

Later descriptions refer to the wrapper classes for a given program as the *program wrapper class*, the *parameter wrapper classes*, the *dynamic array wrapper classes*, and the *substructured-item-array wrapper classes*.

EGL generates a BeanInfo class for each parameter wrapper class, dynamic array wrapper class, or substructured-item-array wrapper class. The BeanInfo class allows the related wrapper class to be used as a Java-compliant Java bean. You probably will not interact with the BeanInfo class.

When you generate a wrapper, the parameter list of the called program cannot include parameters of type BLOB, CLOB, STRING, Dictionary, ArrayDictionary, or non-fixed record.

## Overview of how to use the wrapper classes

To use the wrapper classes to communicate with a program generated with VisualAge Generator, do as follows in the native Java program:

- Instantiate a class (a subclass of CSOPowerServer) to provide middleware services such as converting data between native Java code and a generated program:

```
import com.ibm.javart.v6.cso.*;

public class MyNativeClass
{
    /* declare a variable for middleware */
    CSOPowerServer powerServer = null;

    try
    {
        powerServer = new CSOLocalPowerServerProxy();
    }
    catch (CSOException exception)
    {
        System.out.println("Error initializing middleware"
            + exception.getMessage());
        System.exit(8);
    }
}
```

- Instantiate a program wrapper class to do as follows:
  - Allocate data structures, including dynamic arrays, if any
  - Provide access to methods that in turn access the generated program

The call to the constructor includes the middleware object:

```
myProgram = new MyprogramWrapper(powerServer);
```

- Declare variables that are based on the parameter wrapper classes:

```
Mypart myParm = myProgram.getMyParm();
Mypart2 myParm2 = myProgram.getMyParm2();
```

If your program has parameters that are dynamic arrays, declare additional variables that are each based on a dynamic array wrapper class:

```
myRecArrayVar myParm3 = myProgram.getMyParm3();
```

For details on interacting with dynamic arrays, see “Dynamic array wrapper classes” on page 657.

- In most cases (as in the previous step), use the parameter variables to reference and change memory that was allocated in the program wrapper object
- Set a userid and password, but only in these cases:
  - The Java wrapper accesses an iSeries-based program by way of the iSeries Toolbox for Java; or
  - The generated program runs on a CICS for z/OS region that authenticates remote access.

The userid and password are not used for database access.

You set and review the userid and password by using the callOptions variable of the program object, as in this example:

```
myProgram.callOptions.setUserID("myID");
myProgram.callOptions.setPassword("myWord");
myUserID = myProgram.callOptions.getUserID();
myPassword = myProgram.callOptions.getPassword();
```



- Access the generated program, in most cases by invoking the execute method of the program wrapper object:

```
myProgram.execute();
```

- Use the middleware object to establish database transaction control, but only in the following situation:
  - The program wrapper object either is accessing a generated program on CICS for z/OS or is accessing an iSeries-based COBOL program by way of the IBM Toolbox for Java. In the latter case, the value of **remoteComType** for the call is JAVA400.
  - In the linkage options part used to generate the wrapper classes, you specified that the database unit of work is under client (in this case, wrapper) control; for details, see the reference to **luwControl** in *callLink element*.

If the database unit of work is under client control, processing includes use of commit and rollback methods of the middleware object:

```
powerServer.commit();
powerServer.rollback();
```

- Close the middleware object and allow for garbage collection:

```
if (powerServer != null)
{
    try
    {
        powerServer.close();
        powerServer = null;
    }

    catch(CSOException error)
    {
        System.out.println("Error closing middleware"
            + error.getMessage());
        System.exit(8);
    }
}
```

## The program wrapper class

The program wrapper class includes a private instance variable for each parameter in the generated program. If the parameter is a record or form, the variable refers to an instance of the related parameter wrapper class. If the parameter is a data item, the variable has a primitive Java type.

A table at the end of this help page describes the conversions between EGL and Java types.

A program wrapper object includes the following public methods:

- **get** and **set** methods for each parameter, where the format of the name is as follows:

*purposeParmname()*

*purpose*

The word **get** or **set**

*Parmname*

Name of the data item, record, or form; the first letter is upper case, and aspects of the other letters are determined by the naming convention described in “Naming conventions for Java wrapper classes” on page 659

- An **execute** method for calling the program; you use this method if the data that will be passed as arguments on the call is in the memory allocated for the program wrapper object

Instead of assigning values to the instance variables, you can do as follows:

- Allocate memory for parameter wrapper objects, dynamic array wrapper objects, and primitive types
- Assign values to the memory you allocated
- Pass those values to the program by invoking the **call** method of the program wrapper object rather than the **execute** method

The program wrapper object also includes the `callOptions` variable, which has the following purposes:

- If you generated the Java wrapper so that linkage options for the call are set at generation time, the `callOptions` variable contains the linkage information. For details on when the linkage options are set, see `remoteBind` in *callLink element*.
- If you generated the Java wrapper so that linkage options for the call are set at run time, the `callOptions` variable contains the name of the linkage properties file. The file name is *LO.properties*, where *LO* is the name of the linkage options part used for generation.
- In either case, the `callOptions` variable provides the following methods for setting or getting a userid and password:

```
setPassword(password)
setUserId(userid)
getPassword()
getUserId()
```

The userid and password are used when you set the **remoteComType** property of the `callLink` element to one of the following values:

- CICSECI
- CICSJ2C
- JAVA400

Finally, consider the following situation: your native Java code requires notification when a change is made to a parameter of primitive type. To make such a notification possible, the native code registers as a listener by invoking the **addPropertyChangeListener** method of the program wrapper object. In this case, either of the following situations triggers the `PropertyChange` event that causes the native code to receive notification at run time:

- Your native code invokes a **set** method on a parameter of primitive type
- The generated program returns changed data to a parameter of primitive type

The `PropertyChange` event is described in the JavaBean specification of Sun Microsystems, Inc.

## The set of parameter wrapper classes

A parameter wrapper class is produced for each record that is declared as a parameter in the generated program. In the usual case, you use a parameter wrapper class only to declare a variable that references the parameter, as in the following example:

```
Mypart myRecWrapperObject = myProgram.getMyrecord();
```

In this case, you are using the memory allocated by the program wrapper object.

You also can use the parameter wrapper class to declare memory, as is necessary if you invoke the `call` method (rather than the `execute` method) of the program object.

The parameter wrapper class includes a set of private instance variables, as follows:

- One variable of a Java primitive type for each of the parameter's low-level structure items, but only for a structure item that is neither an array nor within a substructured array
- One array of a Java primitive type for each EGL structure item that is an array and is not substructured
- An object of an inner, array class for each substructured array that is not itself within a substructured array; the inner class can have nested inner classes to represent subordinate substructured arrays

The parameter wrapper class includes several public methods:

- A set of **get** and **set** methods allows you to get and set each instance variable. The format of each method name is as follows:

*purpose*  
*siName*()

*purpose*

The word **get** or **set**.

*siName*

Name of the structure item. The first letter is upper case, and aspects of the other letters are determined by the naming convention described in "Naming conventions for Java wrapper classes" on page 659.

**Note:** Structure items that you declared as fillers are included in the program call; but the array wrapper classes do not include public get or set methods for those structure items.

- The method **equals** allow you to determine whether the values stored in another object of the same class are identical to the values stored in the parameter wrapper object. The method returns **true** only if the classes and values are identical.
- The method **addChangeListener** is invoked if your program requires notification of a change in a variable of a Java primitive type.
- A second set of **get** and **set** methods allow you to get and set the null indicators for each structure item in an SQL record parameter. The format of each of these method names is as follows:

*purpose*  
*siNameNullIndicator*()

*purpose*

The word **get** or **set**.

*siName*

Name of the structure item. The first letter is upper case, and aspects of the other letters are determined by the naming convention described in "Naming conventions for Java wrapper classes" on page 659.

## The set of substructured-item-array wrapper classes

A substructured-item-array wrapper class is an inner class of a parameter class and represents a substructured array in the related parameter. The substructured-item-array wrapper class includes a set of private instance variables that refer to the structure items at and below the array itself:

- One variable of a Java primitive type for each of the array's low-level structure items, but only for a structure item that is neither an array nor within a substructured array

- One array of a Java primitive type for each EGL structure item that is an array and is not substructured
- An object of an inner, substructured-item-array wrapper class for each substructured array that is not itself within a substructured array; the inner class can have nested inner classes to represent subordinate substructured arrays

The substructured-item-array wrapper class includes the following methods:

- A set of **get** and **set** methods for each instance variable

**Note:** Structure items that you declared as nameless fillers are used in the program call; but the substructured-item-array wrapper classes do not include public get or set methods for those structure items.

- The method **equals** allows you to determine whether the values stored in another object of the same class are identical to the values stored in the substructured-item-array wrapper object. The method returns **true** only if the classes and values are identical.
- The method **addPropertyChangeListener**, for use if your program requires notification of a change in a variable of a Java primitive type

In most cases, the name of the top-most substructured-item-array wrapper class in a parameter wrapper class is of the following form:

*ParameterClassname.ArrayClassName*

Consider the following record, for example:

```
Record CompanyPart type basicRecord
10 Departments CHAR(20)[5];
   20 CountryCode CHAR(10);
   20 FunctionCode CHAR(10)[3];
       30 FunctionCategory CHAR(4);
       30 FunctionDetail CHAR(6);
end
```

If the parameter Company is based on CompanyPart, you use the string CompanyPart.Departments as the name of the inner class.

An inner class of an inner class extends use of a dotted syntax. In this example, you use the symbol CompanyPart.Departments.Functioncode as the name of the inner class of Departments.

For additional details on how the substructured-item-array wrapper classes are named, see *Output of Java wrapper generation*.

## Dynamic array wrapper classes

A dynamic array wrapper class is produced for each dynamic array that is declared as a parameter in the generated program. Consider the following EGL program signature:

```
Program myProgram(intParms int[], recParms MyRec[])
```

The name of the dynamic array wrapper classes are IntParmsArray and MyRecArray.

You use a dynamic array wrapper class to declare a variable that references the dynamic array, as in the following examples:

```
IntParmsArray myIntArrayVar = myProgram.getIntParms();
MyRecArray    myRecArrayVar = myProgram.getRecParms();
```

After declaring the variables for each dynamic array, you might add elements:

```
// adding to an array of Java primitives
// is a one-step process
myIntArrayVar.add(new Integer(5));

// adding to an array of records or forms
// requires multiple steps; in this case,
// begin by allocating a new record object
MyRec myLocalRec = (MyRec)myRecArrayVar.makeNewElement();

// the steps to assign values are not shown
// in this example; but after you assign values,
// add the record to the array
myRecArrayVar.add(myLocalRec);

// next, run the program
myProgram.execute();

// when the program returns, you can determine
// the number of elements in the array
int myIntArrayVarSize = myIntArrayVar.size();

// get the first element of the integer array
// and cast it to an Integer object
Integer firstIntElement = (Integer)myIntArrayVar.get(0);

// get the second element of the record array
// and cast it to a MyRec object
MyRec secondRecElement = (MyRec)myRecArrayVar.get(1);
```

As suggested by that example, EGL provides several methods for manipulating the variables that you declared.

Method of the dynamic array class	Purpose
<code>add(int, Object)</code>	To insert an object at the position specified by <i>int</i> and to shift the current and subsequent elements to the right.
<code>add(Object)</code>	To append an object to the end of the dynamic array.
<code>addAll(ArrayList)</code>	To append an <code>ArrayList</code> to the end of the dynamic array.
<code>get()</code>	To retrieve the <code>ArrayList</code> object that contains all elements in the array
<code>get(int)</code>	To retrieve the element that is in the position specified by <i>int</i>
<code>makeNewElement()</code>	To allocate a new element of the array-specific type and to retrieve that element, without adding that element to the dynamic array.
<code>maxSize()</code>	To retrieve an integer that indicates the maximum (but not actual) number of elements in the dynamic array
<code>remove(int)</code>	To remove the element that is in the position specified by <i>int</i>
<code>set(ArrayList)</code>	To use the specified <code>ArrayList</code> as a replacement for the dynamic array
<code>set(int, Object)</code>	To use the specified object as a replacement for the element that is in the position specified by <i>int</i>
<code>size()</code>	To retrieve the number of elements that are in the dynamic array

Exceptions occur in the following cases, among others:

- If you specify an invalid index in the **get** or **set** method
- If you try to add (or set) an element that is of a class incompatible with the class of each element in the array
- If you try to add elements to a dynamic array when the maximum size of the array cannot support the increase; and if the method **addAll** fails for this reason, the method adds no elements

## Naming conventions for Java wrapper classes

EGL creates a name in accordance with these rules:

- If the name is all upper case, lower case all letters.
- If the name is a keyword, precede it with an underline
- If a hyphen or underline is in the name, remove that character and upper case the next letter
- If a dollar sign (\$), at sign (@), or pound sign (#) is in the name, replace each of those characters with a double underscore (\_\_) and precede the name with an underscore (\_).
- If the name is used as a class name, upper case the first letter.

The following rules apply to dynamic array wrapper classes:

- In most cases, the name of a class is based on the name of the part declaration (data item, form, or record) that is the basis of each element in the array. For example, if a record part is called **MyRec** and the array declaration is **recParms myRec[]**, the related dynamic array wrapper class is called **MyRecArray**.
- If the array is based on an item declaration that has no related part declaration, the name of the dynamic array class is based on the array name. For example, if the array declaration is **intParms int[]**, the related dynamic array wrapper class is called **IntParmsArray**.

## Data type cross-reference

The next table indicates the relationship of EGL primitive types in the generated program and the Java data types in the generated wrapper.

EGL primitive type	Length in chars or digits	Length in bytes	Decimals	Java data type	Maximum precision in Java
CHAR	1-32767	2-32766	NA	String	NA
DBCHAR	1-16383	1-32767	NA	String	NA
MBCHAR	1-32767	1-32767	NA	String	NA
UNICODE	1-16383	2-32766	NA	String	NA
HEX	2-75534	1-32767	NA	byte[]	NA
BIN, SMALLINT	4	2	0	short	4
BIN, INT	9	4	0	int	9
BIN, BIGINT	18	8	0	long	18
BIN	4	2	>0	float	4
BIN	9	4	>0	double	15
BIN	18	8	>0	double	15
DECIMAL, PACF	1-3	1-2	0	short	4

EGL primitive type	Length in chars or digits	Length in bytes	Decimals	Java data type	Maximum precision in Java
DECIMAL, PACF	4-9	3-5	0	int	9
DECIMAL, PACF	10-18	6-10	0	long	18
DECIMAL, PACF	1-5	1-3	>0	float	6
DECIMAL, PACF	7-18	4-10	>0	double	15
NUM, NUMC	1-4	1-4	0	short	4
NUM, NUMC	5-9	5-9	0	int	9
NUM, NUMC	10-18	10-18	0	long	18
NUM, NUMC	1-6	1-6	>0	float	6
NUM, NUMC	7-18	7-18	>0	double	15

#### Related concepts

"Java wrapper" on page 390

"Runtime configurations" on page 11

#### Related tasks

"Generating Java wrappers" on page 390

#### Related reference

"callLink element" on page 499

"How Java wrapper names are aliased" on page 776

"Linkage properties file (details)" on page 764

"Output of Java wrapper generation" on page 782

"remoteBind in callLink element" on page 510

## JDBC driver requirements in EGL

The JDBC driver requirements vary by database management system, whether for EGL debug time or run time:

### DB2 UDB

The DB2 Universal driver is compatible with EGL, but the related App driver is not compatible; specifically, the App driver cannot process an EGL **open** or **get** statement that includes the option `forUpdate`.

IBM recommends that you not use the Net driver at all.

If you are running J2EE applications in WebSphere Application Server v6.x, you need DB2 version 8.1.6 or higher. If you are running those applications in WebSphere v5.x Test Environment, you need DB2 version 8.1.3 or higher.

### Informix

The minimum acceptable Informix JDBC driver is 2.21.JC6. This driver level does not comply with JDBC 3.0 and therefore does not support the hold option in the EGL **open** statement. An Informix JDBC 3.0-compliant driver may now be available and should support the hold option.

### Oracle

The JDBC driver that is packaged with Oracle 10i is acceptable.

The following rules apply to any JDBC driver used with EGL:

- The driver must support JDBC 2.0 or higher

- The value `java.sql.ResultSet.CONCUR_UPDATABLE` must be allowed in these contexts:
  - As the second argument to `java.sql.Connection.createStatement(int,int)`
  - As the third argument to `java.sql.Connection.prepareStatement(String,int,int)` and `java.sql.Connection.prepareCall(String,int,int)`
- If you wish to support the hold option in the EGL **open** statement, the driver must support JDBC 3.0, and the value `java.sql.ResultSet.HOLD_CURSORS_OVER_COMMIT` must be allowed in these contexts:
  - As the third argument to `java.sql.Connection.createStatement(int,int,int)`
  - As the fourth argument to `java.sql.Connection.prepareStatement(String,int,int,int)` and `java.sql.Connection.prepareCall(String,int,int,int)`

For any database management system, JDBC drivers from third- or fourth-party vendors are acceptable.

#### Related tasks

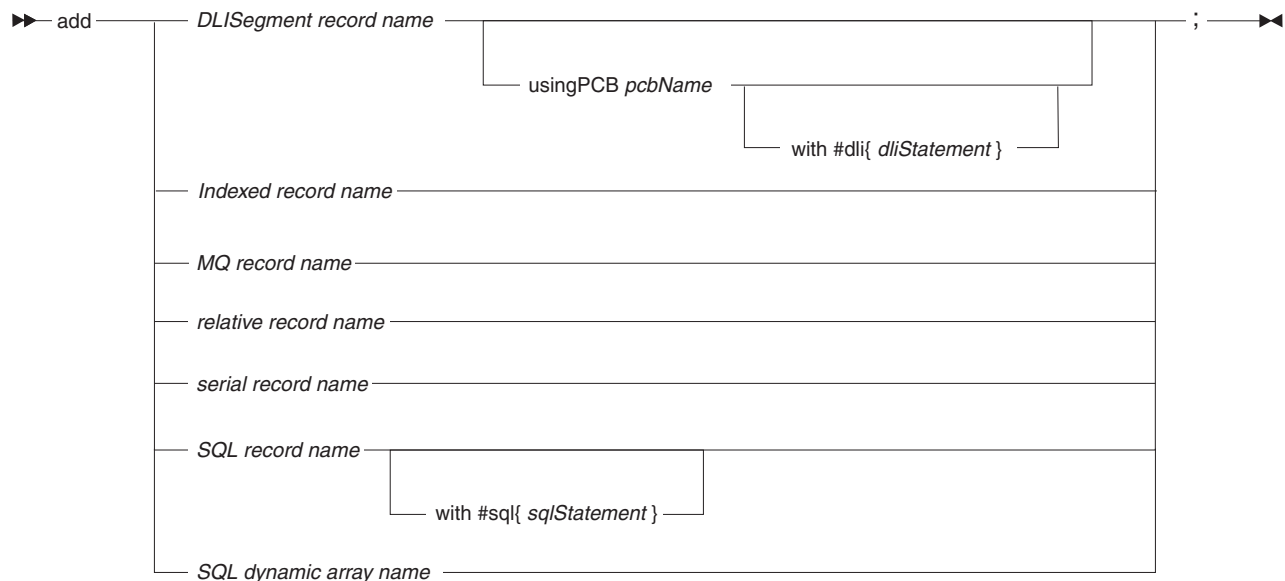
“Setting up a J2EE JDBC connection” on page 445

“Understanding how a standard JDBC connection is made” on page 309

## Keywords

### add

The EGL **add** statement places a record in a file, message queue, or database; or places a set of records in a database.



#### *record name*

Name of the I/O object to add: a DLISegment, indexed, MQ, relative, serial, or SQL record

#### **usingPCB** *pcbName*

Option allowing you to specify the name of a PCB, as defined in your PSB record, to use instead of the default PCB.



**with #dli{** *dliStatement* }

Option that allows an explicit DL/I ISRT call, as described in *DL/I support*.  
Leave no space between #dli and the left brace.

**with #sql{** *sqlStatement* }

Option that allows an explicit SQL INSERT statement. Leave no space after #sql.

*SQL dynamic array name*

The name of a dynamic array of SQL records. The elements are inserted into the database, each at the position specified by the element-specific key values. The operation stops at the first error or when all elements are inserted.

An example is as follows:

```
if (userRequest == "A")
  try
    add record1;
  onException
    myErrorHandler(12);
end
end
```

The behavior of the **add** statement depends on the record type. For details on DL/I processing, see *DLISegment record*. For details on SQL processing, see *SQL record*.

## DLISegment record

The **add** statement generates a DL/I ISRT statement. In DL/I, this insert takes place at the current position in the database—wherever that might be. You may control this positioning by, for example, explicitly setting keys in your EGL program (as shown in the example below), by using a *set record position* statement, by using a **get** statement, or by creating qualified SSAs through the #dli directive.

The following example adds an order to the customer database:

```
//create instances of the records
myCustomer CustomerRecord;
myLocation LocationRecord;
myOrder    OrderRecord;

//build a segment search argument
myCustomer.customerNo = "005001";
myLocation.locationNo = "000022";

//fill the fields in the order record
fillOrder(myOrder);

//add the new order record
try
  add myOrder;
onException
  myErrorHandler(2);
end
```

This **add** statement will generate the following pseudo-DL/I code:

```
ISRT STSCCST (STQCCNO = :myCustomer.customerNo)
      STSCLOC (STQCLNO = :myLocation.locationNo)
      STPCORD
```

Qualified segment search arguments (SSAs) for customer and location identify the parent segments for the new order segment. DL/I will add the new order segment in a position determined by the keyItem field for that segment, the orderDateNo (STQCODN).

To capture "soft" I/O errors like "noRecordFound", use a set-values block when you create the program to set the program property `throwNrfEofExceptions` to yes. In the error handler function, test for the errors with the "is" or the "not" operator:

```
if (myOrder is unique)
  ...
end
```

Possible runtime errors include the following:

- unique if an order segment already exists with the specified key (no duplicates allowed)
- duplicate if an order segment exists with the same key, but duplicates are allowed

DL/I also supports the use of path calls on **add** statements. This means that you can add parent segments for all segment levels between the lowest level segment you are adding and the root. In the following example, DL/I will add a new customer and location at the same time that it adds a new order:

```
add myCustomer, myLocation, myOrder;
```

EGL will generate the following pseudo-DL/I code from this statement:

```
ISRT STSCCST*D (STQCCNO = :myCust.customerNo)
      STSCLOC (STQCLNO = :myLocation.locationNo)
      STPCORD
```

## Indexed record

When you add an indexed record, the key in the record determines the logical position of the record in the file. Adding a record to a file position that is already in use results in the hard I/O error **UNIQUE** or (if duplicates are allowed) in the soft I/O error **DUPLICATE**.

## MQ record

When you add an MQ record, the record is placed at the end of the queue. This placement occurs because the **add** statement invokes one or more MQSeries calls:

- MQCONN connects the generated code to the default queue manager and is invoked when no connection is active
- MQOPEN establishes a connection to the queue and is invoked when a connection is active but the queue is not open
- MQPUT puts the record in the queue and is always invoked unless an error occurred in an earlier MQSeries call

## Relative record

When you add a relative record, the key item specifies the position of the record in the file. Adding a record to a file position that is already in use, however, results in the hard I/O error **UNIQUE**.

The record key item must be available to any function that uses the record and can be any of these:

- An item in the same record
- An item in a record that is global to the program or is local to the function that is running the **add** statement
- A data item that is global to the program or is local to the function that is running the **add** statement

## Serial record

When you add a serial record, the record is placed at the end of the file.

If the generated program adds a serial record and then issues a **get next** statement for the same file, the program closes and reopens the file before executing the **get next** statement. A **get next** statement that follows an **add** statement therefore reads the first record in the file. This behavior also occurs when the **get next** and **add** statements are in different programs, and one program calls another.

It is recommended that you avoid having the same file open in more than one program at the same time.

On CICS for z/OS, a single program cannot include a combination of **add** and **get next** statements for the same spool file. This restriction also applies when the **add** and **get next** statements are in different programs, and one program calls the other. Similarly, an **add** that follows a **get** or **get next** statement will add a record to the beginning of the file.

For IMS BMP or z/OS batch, if you add a variable-length serial record to a file associated with GSAM and the record length is longer than the physical file, DL/I returns a blank status code. Data is truncated, but no message is issued because the situation cannot be detected.

For IMS/VS, you must associate a serial record with an alternate PCB (a TP PCB in the PSB). The IMS message header (length, ZZ field, and transaction code) is automatically added to each record written to the message queue. An EGL **add** statement for a serial record assigned to a message queue results in an ISRT call to the message queue. If an error occurs and the record is assigned to a multiple segment message queue and associated with the express alternate PCB, any records already added are committed, even if an explicit **close** statement has not occurred. If it is important that these records are not committed, include an additional express alternate PCB in the PSB and associate the file with the additional express alternate PCB.

## SQL record

Some error conditions are as follows:

- You specify an SQL statement of a type other than INSERT
- You specify some but not all clauses of an SQL INSERT statement
- You specify an SQL INSERT statement (or accept an implicit SQL statement) that has any of these characteristics--
  - Is related to more than one SQL table
  - Includes only host variables that you declared as read only
  - Is associated with a column that either does not exist or is incompatible with the related host variable

The result is as follows when you add an SQL record without specifying an explicit SQL statement:

- The format of the generated SQL INSERT statement looks like this:

```
INSERT INTO tableName
  (column01, ... columnNN)
values (:recordItem01, ... :recordItemNN)
```

- The key value in the record determines the logical position of the data in the table. A record that does not have a key is handled in accordance with the SQL table definition and the rules of the database.

- As a result of the association of record items and SQL table columns in the record part, the generated code places the data from each record item into the related SQL table column.
- If you declared a record item to be read only, the generated SQL INSERT statement does not include that record item, and the database management system sets the value of the related SQL table column to the default value that was specified when the column was defined.

An example that uses a dynamic array of SQL records is as follows:

```
try
  add employees;
onException
  sysLib.rollback();
end
```

### Related concepts

“DL/I database support” on page 310

“References to parts” on page 23

“Record types and properties” on page 138

“SQL support” on page 277

### Related tasks

“Syntax diagram for EGL statements and commands” on page 884

**Related reference** “#dli directive” on page 325

“CICS-related considerations” on page 530

“close” on page 669

“delete” on page 673

“get” on page 687

“get next” on page 701

“get” on page 687

“get previous” on page 708

“Exception handling” on page 94

“execute” on page 677

“I/O error values” on page 638

“open” on page 722

“prepare” on page 736

“EGL statements” on page 88

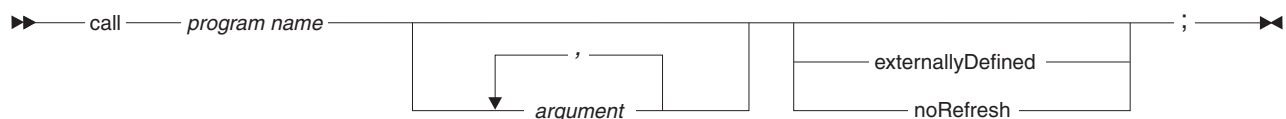
“replace” on page 738

“set” on page 742

“SQL item properties” on page 68

## call

The EGL call statement transfers control to another program and optionally passes a series of values. Control returns to the caller when the called program ends; and if the called program changes any data that was passed by way of a variable, the storage area available to the caller is changed, too.



**program name**

Name of the called program. The program is either generated by EGL or is considered *externally defined*.

The specified name cannot be a reserved word. If the caller must call a non-EGL program that has the same name as an EGL reserved word, use a different program name in the call statement, then use a linkage options part, **callLink** element to specify an alias, which is the name used at run time.

If the called program is a Java program, the called program name is case-sensitive; *calledProgram* is different from *CALLEDPROGRAM*. Otherwise, the determination of whether the program name is case-sensitive depends on the system on which the called program resides: case-sensitive for UNIX, case-insensitive otherwise.

In the EGL debugger, the called program name is case-insensitive.

**argument**

One of a series of value references, each separated from the next by a comma. An argument may be a primitive variable; a form; a record; an array of primitive types; an array of records (of type *BasicRecord*, *DLISegment*, or *SQLRecord*); a non-numeric literal; a non-numeric constant; or (if EGL has access to the called program at generation time) a more complex datetime, numeric, or text expression. You may not pass a field of type *ANY*, *ArrayDictionary*, *Blob*, *Clob*, *DataTable*, or *Dictionary*. Also, you may not pass arrays of those types or records that include any of those types.

**externallyDefined**

An indicator that the program is externally defined. This indicator is available only if you set the project property for VisualAge Generator compatibility.

It is recommended that a non-EGL-generated program be identified as externally defined not in the **call** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, **callLink** element, and is also called **externallyDefined**.)

**noRefresh**

An indicator that a screen refresh is to be avoided when the called program returns control.

The indicator is supported (at development time) if the program property **VAGCompatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

This indicator is appropriate if the caller is in a run unit that presents text forms to a screen and either of these situations is in effect:

- The called program does not present a text form; or
- The caller writes a full-screen text form after the call.

It is recommended that you indicate your preference for screen refresh not in the **call** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, **callLink** element, and is called **refreshScreen**.)

An example is as follows:

```

if (userRequest == "C")
  try
    call programA;
  onException
    myErrorHandler(12);
end
end

```

The number, type, and sequence of arguments in a call statement must correspond to the number, type, and sequence of values expected by the called program.

It is strongly recommended that the number of bytes passed in each argument be the same as the number of bytes expected. If the called program is in CICS and the lengths are unequal, a runtime abend occurs. In the case of an EGL-generated Java program, a length mismatch causes an error only if the runtime correction of that mismatch causes a type mismatch:

- If the called Java program receives too few bytes, the end of the passed data is padded with blanks.
- If the called Java program receives too many bytes, the end of the passed data is truncated.

In the case of Java, an error occurs if blanks are added to a data item of type NUM, for example, but not if blanks are added to a data item of type CHAR.

The following rules apply to literals and constants:

- The size of a passed literal or constant must equal the size of the receiving parameter
- A numeric literal or constant cannot be passed as an argument
- A literal or constant that includes only single-byte characters may be passed to a parameter of type CHAR or MBCHAR
- A literal or constant that includes only double-byte characters may be passed only to a parameter of type DBCHAR
- A literal or constant that includes a combination of single- and double-byte characters may be passed to a parameter of type MBCHAR

Recursive calls are supported when you generate Java code, but not when you generate COBOL programs.

The call is affected by the linkage options part, if any, that is used at generation time. (You include a linkage options part by setting the build descriptor option **linkage**.)

For details on linkage, see *Linkage options part*.

### **Related concepts**

"Linkage options part" on page 399

"Syntax diagram for EGL statements and commands" on page 884

### **Related reference**

"EGL statements" on page 88

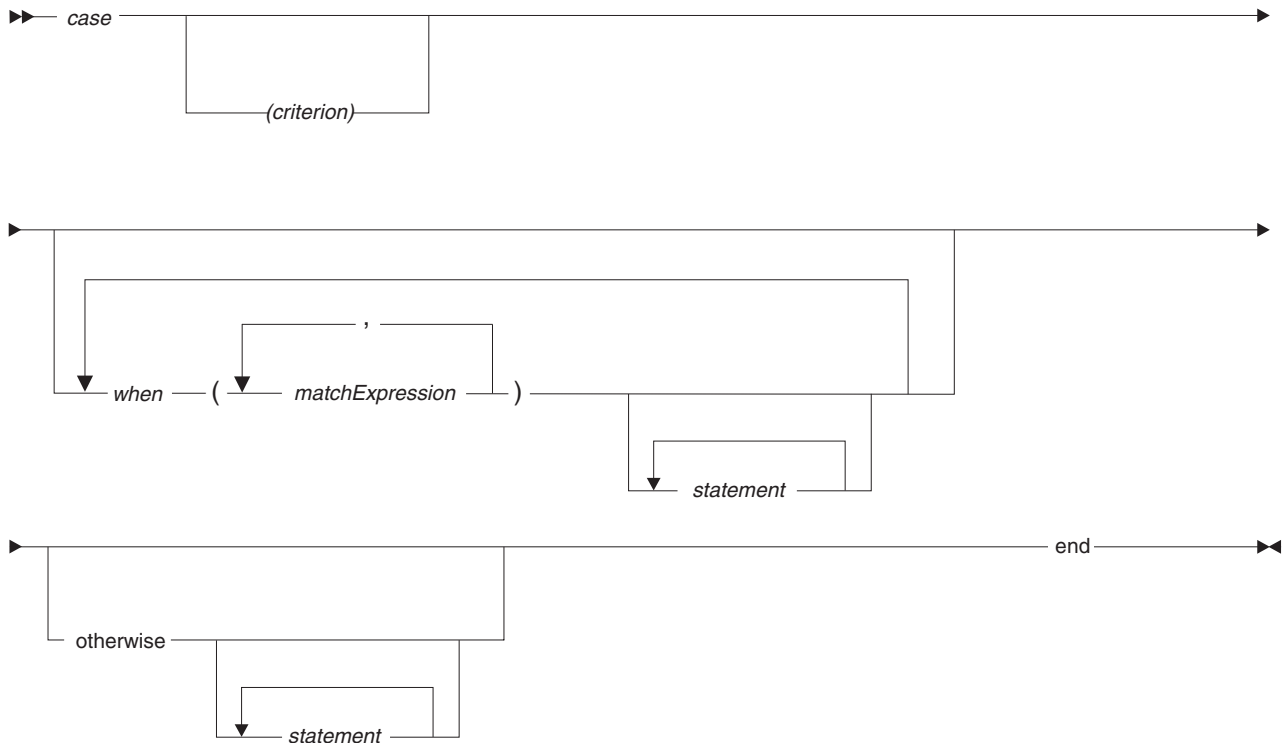
"Exception handling" on page 94

"linkage" on page 484

"Primitive types" on page 34

## case

The EGL **case** statement marks the start of multiple sets of statements, where at most only one of those sets is run. The **case** statement is equivalent to a C or Java **switch** statement that has a break at the end of each case clause.



### *criterion*

An item, constant, expression, literal, or system variable, including `ConverseVar.eventKey` or `sysVar.systemType`.

If you specify *criterion*, each of the subsequent **when** clauses (if any) must contain one or more instances of *matchExpression*. If you do not specify *criterion*, each of the subsequent **when** clauses (if any) must contain a *logical expression*.

### **when**

The beginning of a clause that is invoked only in these cases:

- You specified a *criterion*, and the **when** clause is the first to contain a *matchExpression* that is equal to the *criterion*; or
- You did not specify a *criterion*, and the **when** clause is the first to contain a *logical expression* that evaluates to true.

If you wish the **when** clause to have no effect when invoked, code the clause without EGL statements.

A **case** statement may have any number of **when** clauses.

### *matchExpression*

One of the following values:

- A numeric or string expression
- A symbol for comparison to `ConverseVar.eventKey` or `sysVar.systemType`

The primitive type of *matchExpression* value must be compatible with the primitive type of the *criterion* value. For details on compatibility, see *Logical expressions*.

*logicalExpression*  
A logical expression.

*statement*  
An EGL statement.

### **otherwise**

The beginning of a clause that is invoked if no when clause runs.

After the statements run in a when or otherwise clause, control passes to the EGL statement that immediately follows the end of the **case** statement. Control does not fall through to the next when clause under any circumstance. If no when clause is invoked and no default clause is in use, control also passes to the next statement immediately following the end of the **case** statement, and no error situation is in effect.

An example of a **case** statement is as follows:

```
case (myRecord.requestID)
  when (1)
    myFirstFunction();
  when (2, 3, 4)
    try
      call myProgram;
    onException
      myCallFunction(12);
    end
  otherwise
    myDefaultFunction();
end
```

If a single clause includes multiple instances of *matchExpression* (2, 3, 4 in the previous example), evaluation of those instances is from left to right, and the evaluation stops as soon as one *matchExpression* is found that corresponds to the criterion value.

### **Related tasks**

“Syntax diagram for EGL statements and commands” on page 884

### **Related reference**

“EGL statements” on page 88

“Logical expressions” on page 593

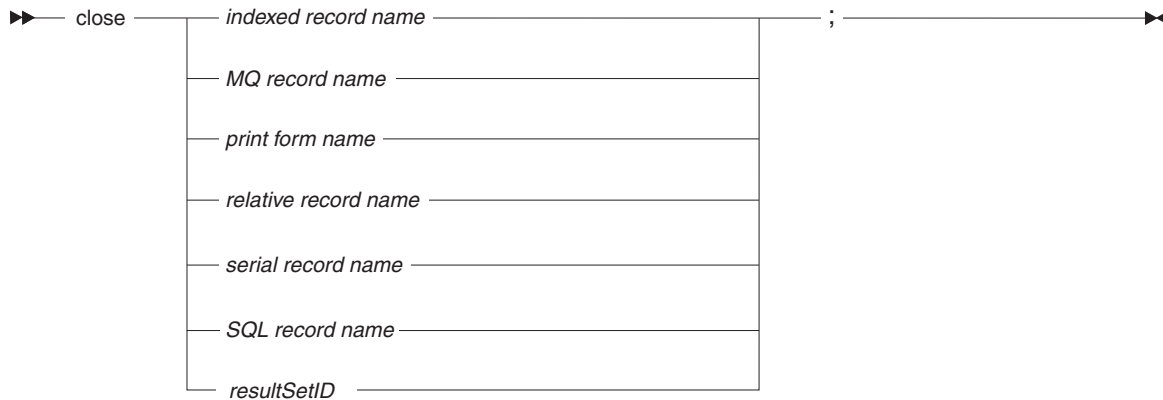
“eventKey” on page 1058

“systemType” on page 1074

## **close**

The EGL **close** statement disconnects a printer; or closes the file or message queue associated with a given record; or, in the case of an SQL record, closes the cursor that was opened by an EGL **open** or **get** statement.





*name*

Name of the I/O object that is associated with the resource being closed; that object is a print form or an indexed, MQ, relative, serial, or SQL record

*resultSetIdentifier*

For SQL processing only, an ID that ties the **close** statement to a **get** or **open** statement run earlier in the same program. For details, see *resultSetID*.

Example:

```

if (userRequest == "C")
  try
    close fileA;
  onException
    myErrorHandler(12);
  end
end
  
```

The behavior of a **close** statement depends on the type of I/O object that is associated with the resource being closed.

### Indexed, serial, or relative record

When you use the name of an indexed, serial, or relative record in a **close** statement, EGL closes the file associated with that record.

If a file is open and you use the fileAssociation item to change the resource name associated with that file, EGL closes the file automatically before executing the next statement that affects the file. For details, see *resourceAssociation*.

EGL also closes any file that is open when the program ends.

On CICS for z/OS, these rules apply:

- If you close a serial record associated with a spool file, EGL ensures that the CICS SPOOL CLOSE command is executed for that file
- If you close a serial record that is not associated with a spool file or if you close an indexed or relative record, EGL performs a logical close on the file, which is then accessible by other programs
- A **close** statement does not delete temporary storage files

### MQ record

When you use the name of a MQ record in a **close** statement, EGL ensures that the MQSeries command MQCLOSE is executed for the message queue associated with that record.

## Print form

If the I/O object is a print form, the close statement issues a form feed and either disconnects from the printer or (if the print form is spooled to a file) closes the file.

Before you use `ConverseVar.printerAssociation` to change the print destination, close the printer or file specified by the current value of `ConverseVar.printerAssociation`. Issue a close statement option for each print destination, as multiple printer or print files can be open at the same time.

The EGL runtime ensures that all printers are closed when the program ends.

## SQL record

When you use the name of an SQL record in a **close** statement, EGL closes the SQL cursor that is open for that record.

EGL automatically closes a cursor in these cases:

- A cursor-processing loop follows an **open** statement and continues until a No Record Found (NRF) condition indicates that all rows in the set were processed
- EGL runs a **get** statement for an SQL record when a single row is read and neither `forUpdate` nor `singleRow` was specified as an option
- EGL runs a **replace** or **delete** statement that uses the cursor opened by a **get** statement; in this case, `forUpdate` was specified as an option in the **get** statement
- EGL begins to process an **open** or **get** statement for a record that is associated with an open cursor; the close precedes the other processing
- The program runs either **sysLib.commit** or **sysLib.rollback**; but the close does not occur if the option **hold** is in effect, as explained in relation to open

EGL closes all open cursors in this case:

- The program is of type `VGWebTransaction` and presents a Web page
- The program is of type `textUI`, does an automatic commit before conversing a form, and is unaffected by the option **hold** when the converse occurs; for details on `textUI` programs and the **converse** statement, see *Segmentation*

## Related concepts

"Record types and properties" on page 138

"resultSetID" on page 867

"Segmentation in text applications" on page 189

"SQL support" on page 277

## Related tasks

"Syntax diagram for EGL statements and commands" on page 884

## Related reference

"add" on page 661

"delete" on page 673

"EGL statements" on page 88

"Exception handling" on page 94

"execute" on page 677

"get" on page 687

"get next" on page 701

"get previous" on page 708

"I/O error values" on page 638

"open" on page 722

"prepare" on page 736

“replace” on page 738  
 “recordName.resourceAssociation” on page 985  
 “SQL item properties” on page 68  
 “commit()” on page 1024  
 “rollback()” on page 1036  
 “printerAssociation” on page 1059  
 “terminalID” on page 1075

## continue

The EGL **continue** statement transfers control to the end of a **for**, **forEach**, **openUI**, or **while** statement that itself contains the **continue** statement. Execution of the containing statement continues or ends depending on the logical test that is conducted as usual at the start of the containing statement.

The **continue** statement must be in the same function as the containing statement.



### for, forEach, or while

Identifies the innermost containing statement of the specified type. If you specify one of those statement types, the **continue** statement must be contained in a statement of that type. If you do not specify a statement type, the result is as follows:

- The **continue** statement transfers control to the end of the innermost containing **for**, **forEach**, or **while** statement; and
- The **continue** statement must be contained in a statement of one of those types.

### Related tasks

“Syntax diagram for EGL statements and commands” on page 884

### Related reference

“EGL statements” on page 88

## converse

The EGL **converse** statement presents a text form in a text application or presents a VGUI record in a program of type VGWebtransaction.

The program waits for a user response, receives the text form or VGUI record from the user, and continues processing with the statement that follows the **converse** statement.

For an overview of text-form processing, see the following pages in order:

1. *Text forms*

## 2. Segmentation

For an overview of Web processing, see the following pages in order:

1. *Web transaction support in EGL*
2. *Segmentation in Web applications*



### *textFormName*

Name of a text form that is visible to the program. For details on visibility, see *References to parts*.

### *VGUIRecordName*

The name of a VGUI record that is visible to the program. For details on that record, see *VGUIRecord part in EGL source format*. For details on visibility, see *References to parts*.

An example is as follows:

```
converse myTextForm;
```

These considerations apply:

- In relation to text forms, a **converse** statement is always valid in a called program; but if you are running a main program that is segmented, the **converse** statement is not valid in these kinds of code--
  - A function that has parameters, local storage, or return values
  - A function that is invoked (directly or indirectly) by a function that has parameters, local storage, or return values.
- Although use of a **converse** statement is relatively simple, you get better performance by using a **show** statement that returns to the beginning of the same program. Use of a **show** statement requires a more complicated design because the re-invoked program starts at the first line, and that initial code must analyze whether the program is being invoked at the beginning or in the middle of a user-code interaction.

### **Related concepts**

“References to parts” on page 23

“Segmentation in text applications” on page 189

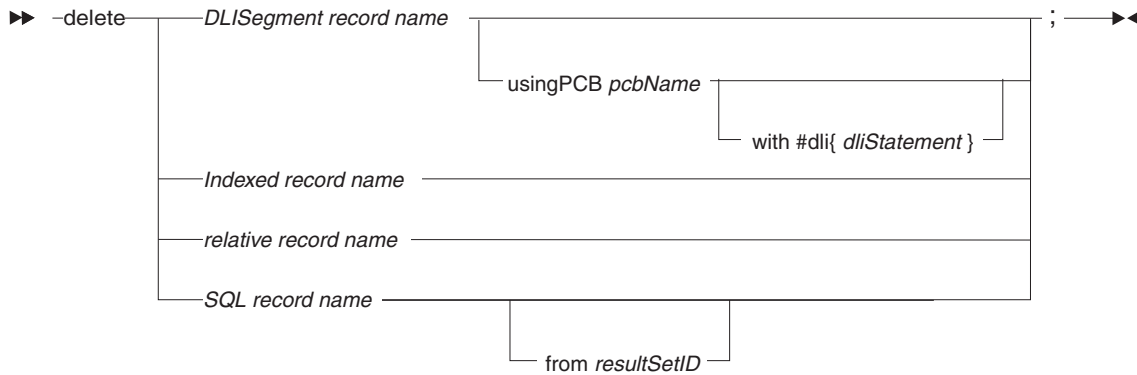
“Segmentation in Web transactions” on page 250

“Text forms” on page 188

“Web transaction support in EGL” on page 163

## **delete**

The EGL **delete** statement removes either a record from a file or a row or a segment from a DL/I database.



#### *record name*

Name of the I/O object: a DLISegment, indexed, relative, or SQL record being deleted

#### **usingPCB** *pcbName*

Option allowing you to specify the name of a PCB, as defined in your PSB record, to use instead of the default PCB.

#### **with #dli{** *dliStatement* **}**

Option that allows an explicit DL/I DLET call, as described in *#dli directive*. Leave no space between **#dli** and the left brace.

#### **from** *resultSetID*

ID that ties the **delete** statement to a **get** or **open** statement run earlier in the same program. For details, see *resultSetID*.

An example is as follows:

```

if (userRequest == "D")
  try
    get myRecord forUpdate;
    onException
      myErrorHandler(12); // exits the program
  end

  try
    delete myRecord;
    onException
      myErrorHandler(16);
  end
end

```

The behavior of the **delete** statement depends on the record type. For details on DL/I processing, see *DLISegment record*. For SQL processing, see *SQL record*.

## **DLISegment record**

The **delete** statement generates a DL/I DLET statement. In DL/I, you must get and hold a segment before deleting it. The EGL keywords **get...forUpdate**, **get next...forUpdate**, and **get next inParent...forUpdate** will all hold the requested segment for deletion.

The following example deletes an order from the customer database:

```

//create instances of the records
myCustomer CustomerRecord;
myLocation LocationRecord;
myOrder OrderRecord;

//build a segment search argument

```

```

myCustomer.customerNo = "005001";
myLocation.locationNo = "000022";
myOrder.orderDateNo = "20050730A003";

//hold the requested order record
try
  get myOrder forUpdate;
  onException
    myErrorHandler(2);
end

//delete the order
try
  delete myOrder;
  onException
    myErrorHandler(7);
end

```

The **get...forUpdate** statement generates qualified SSAs for customer, location, and order:

```

GHU STSCCST (STQCCNO = :myCust.customerNo)
      STSCLOC (STQCLNO = :myLocation.locationNo)
      STPCORD (STQCODN = :myOrder.orderDateNo)

```

The **delete** statement then simply generates the following:

```

DLET STPCORD

```

## Indexed or relative record

If you want to delete an indexed or relative record, do as follows:

- Issue a **get** statement for the record and specify the **forUpdate** option
- Issue the **delete** statement, with no intervening I/O operation against the same file

After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** on the same file (with the **forUpdate** option), a subsequent **replace** or **delete** is valid on the newly read file record
- If the next I/O operation is a **get** on the same EGL record (with no **forUpdate** option) or is a **close** on the same file, the file record is released without change

For details on the **forUpdate** option, see *get*.

## SQL record

In the case of SQL processing, you must use the **forUpdate** option on an EGL **get** or **open** statement to retrieve a row for subsequent deletion:

- You can issue a **get** statement to retrieve the row; or
- You can issue an **open** statement to select a set of rows and then invoke a **get next** statement to retrieve the row of interest.

In either case, the EGL **delete** statement is represented in the generated code by an SQL DELETE statement that references the current row in a cursor. You cannot modify that SQL statement, which is formatted as follows:

```
DELETE FROM tableName
WHERE CURRENT OF cursor
```

If you wish to write your own SQL DELETE statement, use the EGL **execute** statement.

You cannot use a single EGL **delete** statement to remove rows from multiple SQL tables.

#### Related concepts

“DL/I database support” on page 310  
“Record types and properties” on page 138  
“resultSetID” on page 867  
“Run unit” on page 866  
“SQL support” on page 277

#### Related tasks

“Syntax diagram for EGL statements and commands” on page 884

#### Related reference

“#dli directive” on page 325  
“add” on page 661  
“CICS-related considerations” on page 530  
“close” on page 669  
“EGL statements” on page 88  
“Exception handling” on page 94  
“execute” on page 677  
“get” on page 687  
“get next” on page 701  
“get” on page 687  
“get previous” on page 708  
“I/O error values” on page 638  
“prepare” on page 736  
“open” on page 722  
“replace” on page 738  
“set” on page 742  
“SQL item properties” on page 68  
“terminalID” on page 1075

## display

The EGL **display** statement adds a text form to a runtime buffer but does not present data to the screen. For details on the runtime behavior, see *Text forms*.

**Note:** If you are working in VisualAge Generator compatibility mode, you can issue a statement of the following form:

```
display printForm;  
printForm  
    Name of a print form that is visible to the program.
```

In that case, **display** is equivalent to **print**.

► — display — *textFormName* ————— ; ————— ◀◀

**textFormName**

Name of a text form that is visible to the program. For details on visibility, see *References to parts*.

**Related concepts**

"References to parts" on page 23

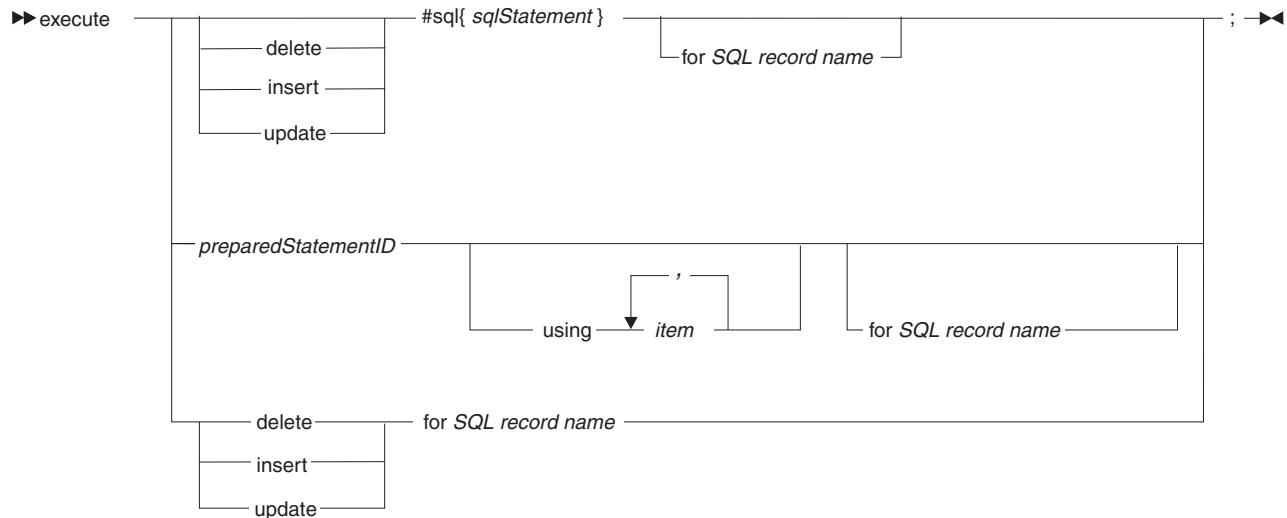
"Text forms" on page 188

**Related reference**

"print" on page 737

**execute**

The EGL **execute** statement lets you write one or more SQL statements; in particular, SQL data-definition statements (of type CREATE TABLE, for example) and data-manipulation statements (of type INSERT or UPDATE, for example)

**#sql{ sqlStatement }**

An explicit SQL statement. If you want the SQL statement to update or delete a row in a result set, code an SQL UPDATE or DELETE statement that includes the following clause:

WHERE CURRENT OF *resultSetID*

**resultSetID**

The resultSetID specified in the EGL open statement that made the result set available.

Leave no space between #sql and the left brace.

**for SQL record name**

Name of an SQL record.

If you specify a statement type (delete, insert, or update), EGL uses the SQL record to build an implicit SQL statement, as described later. In any case, you can use the SQL record to test the outcome of the operation.

**preparedStatementID**

Refers to an EGL prepare statement that has the specified ID. If you do not



reference a prepare statement, you must specify either an explicit SQL statement or a combination of an SQL record and a statement type (delete, insert, or update).

#### **delete, insert, update**

Indicates that EGL is to provide an implicit SQL statement of the specified type. A declaration-time error occurs if you specify a statement type but not an SQL record name.

If you do not set a statement type, you must specify either an explicit SQL statement or a reference to a prepare statement.

For an overview of implicit SQL statements, see *SQL support*.

Several example statements are as follows (assuming that `employeeRecord` is an SQL record):

```
execute
  #sql{
    create table employee (
      empnum decimal(6,0) not null,
      empname char(40) not null,
      empphone char(10) not null)
  };

execute update for employeeRecord;

execute
  #sql{
    call aStoredProcedure( :argumentItem)
  };
```

You can use an **execute** statement to issue SQL statements of the following types:

- ALTER
- CALL
- CREATE ALIAS
- CREATE INDEX
- CREATE SYNONYM
- CREATE TABLE
- CREATE VIEW
- DECLARE global temporary table
- DELETE
- DROP INDEX
- DROP SYNONYM
- DROP TABLE
- DROP VIEW
- GRANT
- INSERT
- LOCK
- RENAME
- REVOKE
- SAVEPOINT
- SET
- SIGNAL
- UPDATE
- VALUES

You cannot use an **execute** statement to issue SQL statements of the following types:

- CLOSE

- COMMIT
- CONNECT
- CREATE FUNCTION
- CREATE PROCEDURE
- DECLARE CURSOR
- DESCRIBE
- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- OPEN
- PREPARE
- ROLLBACK WORK
- SELECT
- INCLUDE SQLCA
- INCLUDE SQLDA
- WHENEVER

### Implicit SQL DELETE

The effect of requesting an implicit SQL DELETE statement is that an SQL record property (**defaultSelectCondition**) determines what table rows are deleted, so long as the value in each SQL table key column is equal to the value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are deleted.

The implicit SQL DELETE statement for a particular record is similar to the following statement:

```
DELETE FROM tableName
WHERE keyColumn01 = :keyItem01
```

You cannot use a single EGL statement to delete rows from more than one database table.

### Implicit SQL INSERT

The effect of requesting an implicit SQL INSERT statement is as follows by default:

- The key value in the record determines the logical position of the data in the table. A record that does not have a key is handled in accordance with the SQL table definition and the rules of the database.
- As a result of the association of record items and SQL table columns in the record part, the generated code places the data from each record item into the related SQL table column.
- If you declared a record item to be read only, the generated SQL INSERT statement does not include that record item, and the database management system sets the value of the related SQL table column to the default value that was specified when the column was defined.

The format of the implicit SQL INSERT statement is like this:

```
INSERT INTO tableName
(column01, ... columnNN)
values (:recordItem01, ... :recordItemNN)
```

Some error conditions are as follows:

- You specify an SQL statement of a type other than INSERT
- You specify some but not all clauses of an SQL INSERT statement

- You specify an SQL INSERT statement (or accept an implicit SQL statement) that has any of these characteristics--
  - Is related to more than one SQL table
  - Includes only host variables that you declared as read only
  - Is associated with a column that either does not exist or is incompatible with the related host variable

## Implicit SQL UPDATE

The effect of requesting an implicit SQL UPDATE statement is as follows by default:

- An SQL record property (**defaultSelectCondition**) determines what table rows are selected, so long as the value in each SQL table key column is equal to the value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are updated.
- As a result of the association of record items and SQL table columns in the SQL record declaration, a given SQL table column receives the content of the related record item. If an SQL table column is associated with a record item that is read only, however, that column is not updated.

The format of the implicit SQL UPDATE statement for a particular record is similar to the following statement:

```
UPDATE tableName
SET   column01 = :recordItem01,
      column02 = :recordItem01, ...
      columnNN = :recordItemNN
WHERE keyColumn01 = :keyItem01
```

An error occurs in any of the following cases:

- All the items are identified as read only
- The statement attempts to update more than one SQL table
- An item whose value is being written to the database is associated with a column that either does not exist at run time or is incompatible with that item

## Related concepts

“Record types and properties” on page 138

“SQL support” on page 277

“References to parts” on page 23

## Related tasks

“Syntax diagram for EGL statements and commands” on page 884

## Related reference

“add” on page 661

“close” on page 669

“delete” on page 673

“EGL statements” on page 88

“Exception handling” on page 94

“get” on page 687

“get next” on page 701

“get previous” on page 708

“I/O error values” on page 638

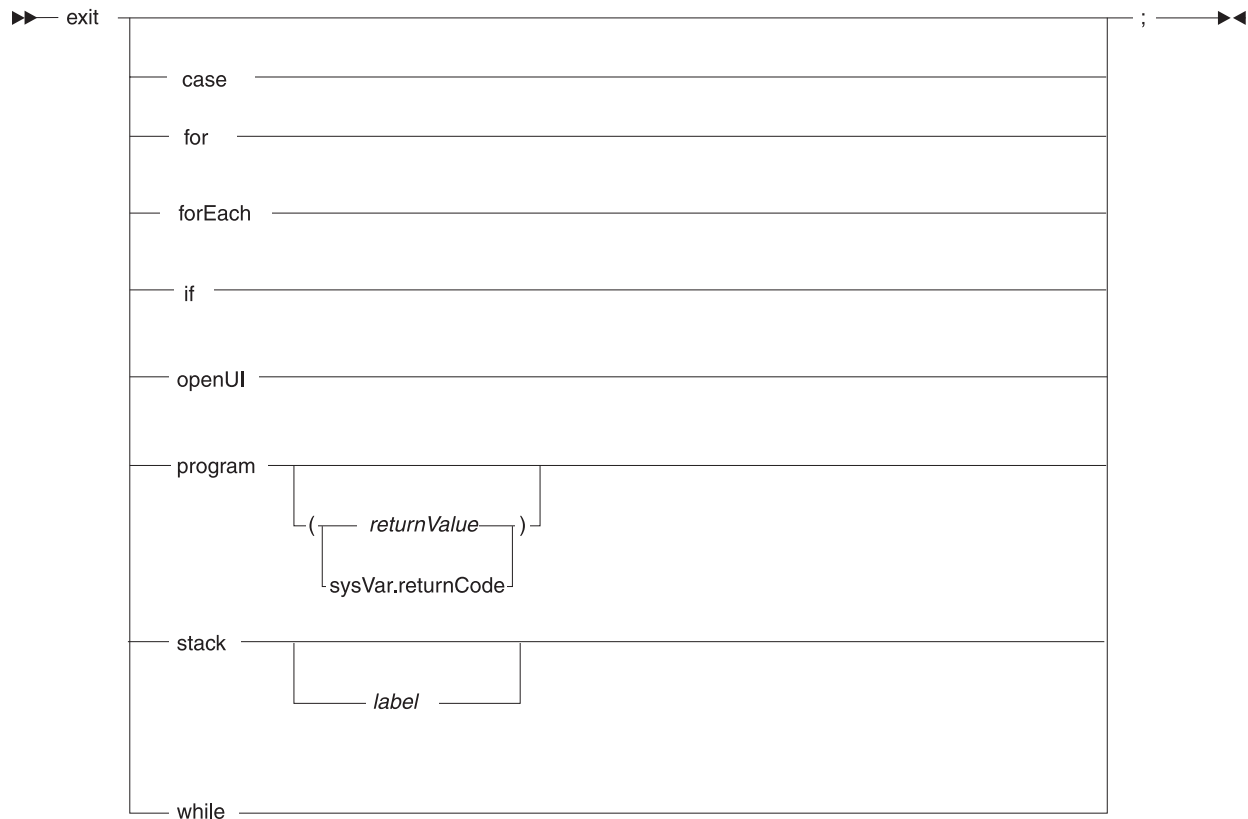
“open” on page 722

“prepare” on page 736

[“replace” on page 738](#)  
[“SQL item properties” on page 68](#)  
[“terminalID” on page 1075](#)

## exit

The EGL **exit** statement leaves the specified block, which by default is the block that immediately contains the **exit** statement.



### case

Leaves the most recently entered **case** statement in which the **exit** statement resides. Continues processing after the **case** statement.

An error occurs if the **exit** statement is not inside a **case** statement that begins in the same function.

### for

Leaves the most recently entered **for** statement in which the **exit** statement resides. Continues processing after the **for** statement.

An error occurs if the **exit** statement is not inside a **for** statement that begins in the same function.

### forEach

Leaves the most recently entered **forEach** statement in which the **exit** statement resides. Continues processing after the **forEach** statement.

An error occurs if the **exit** statement is not inside a **forEach** statement that begins in the same function.

**if** Leaves the most recently entered **if** statement in which the **exit** statement resides. Continues processing after the **if** statement.

An error occurs if the **exit** statement is not inside an **if** statement that begins in the same function.

**program**

Leaves the program.

The value in the system variable **sysVar.returnValue** is returned to the operating system in any of the following cases:

- The program ends with an **exit** statement that does not include a return code
- The program ends with an **exit** statement that returns **sysVar.returnValue**
- The program ends without a terminating **exit** statement

If the program ends with a terminating **exit** statement that includes a return code other than **sysVar.returnValue**, the specified value is used in place of any value that may be in **sysVar.returnValue**.

*returnValue*

A literal integer or an item, constant, or numeric expression that resolves to an integer. The return value is made available to the operating system. For Java output, the value must be in the range of -2147483648 to 2147483647, inclusive. For COBOL output, the value must be in the range of 0 to 512, inclusive.

For other details on return values, see *sysVar.returnValue*.

**sysVar.returnValue**

The system variable that includes the value returned to the operating system.

For details, see *sysVar.returnValue*.

**stack**

Returns control to the main function without setting a return value for the current function.

A statement of the form *exit stack* removes all references to the intermediate functions in the runtime *stack*, which is a list of functions; specifically, the current function plus the series of functions whose running made possible the running of the current function.

The main function may have invoked a function (now in the stack), and the invocation may have included a parameter that had the modifier **out** or **inOut**. In those cases, the **exit** statement of the form *exit stack* makes the value of the parameters available to the main function.

If you do not specify a *label* (as described later), processing continues at the statement after the most recently run function invocation in the main function. If you specify a label, processing continues at the statement that follows the label in the main function. The label may precede or follow the most recently run function invocation in the main function.

If you specify an **exit** statement of the form *exit stack* in the main function, the next statement is processed, even if you specify a label. For details on how to go to a specified label in the current function, see *goTo*.

*label*

A series of characters that are displayed in the main function and outside of any blocks, including these:

- **if**

- else
- inside a **case** statement
- while
- try

When displayed at the location where processing continues, the label is followed by colon. For details on valid characters for the label, see *Naming conventions*.

#### Related reference

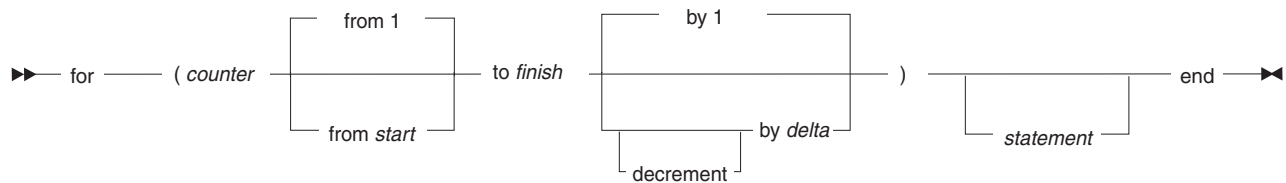
“goTo” on page 714

“Naming conventions” on page 778

“returnCode” on page 1070

## for

The EGL keyword **for** begins a statement block that runs in a loop for as many times as a test evaluates to true. The test is conducted at the beginning of the loop and indicates whether the value of a counter is within a specified range. The keyword **end** marks the close of the **for** statement.



#### counter

A numeric variable without decimal places. EGL statements in the **for** statement can change the value of *counter*.

#### from start

The initial value of *counter*. The initial value is 1 if you do not specify a clause that begins with **from**.

*start* can be any of these:

- An integer literal
- A numeric variable without decimal places
- A numeric expression, which must resolve to an integer

#### to finish

If you do not specify **decrement**, *finish* is the upper limit of *counter*; and if the value of *counter* exceeds that limit, the test mentioned earlier resolves to false, the statement block is no longer executed, and the **for** statement ends.

If you specify **decrement**, *finish* is the lower limit of *counter*; and if the value of *counter* is below that limit, the test resolves to false, the statement block is no longer executed, and the **for** statement ends.

*finish* can be any of these:

- An integer literal
- A numeric variable without decimal places
- A numeric expression, which must resolve to an integer

EGL statements in the **for** statement can change the value of *finish*.

### by *delta*

If you do not specify **decrement**, *delta* is the value added to *counter* after the EGL statement block is executed and before the value of *counter* is tested.

If you specify **decrement**, *delta* is the value subtracted from *counter* after the EGL statement block is executed and before the value of *counter* is tested.

*delta* can be any of these:

- An integer literal
- A numeric variable without decimal places
- A numeric expression, which must resolve to an integer

EGL statements in the **for** statement can change the value of *delta*.

### *statement*

A statement in the EGL language

An example is as follows:

```
sum = 0;

// adds 10 values to sum
for (i from 1 to 10 by 1)
    sum = inputArray[i] + sum;
end
```

### Related reference

"EGL statements" on page 88

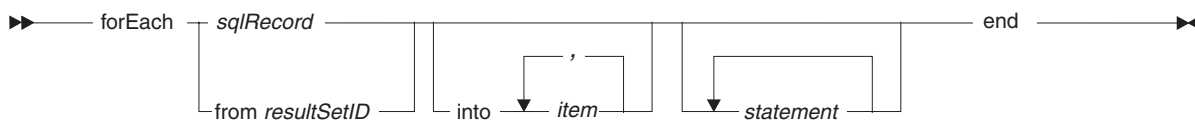
### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

## forEach

The EGL keyword **forEach** marks the start of a set of statements that run in a loop. The first iteration occurs only if a specified result set is available. (If the result set is not available, the statement fails with a hard error.) The loop reads each row in the result set, continuing until one of these events occurs:

- All rows are retrieved;
- An **exit** statement runs; or
- A hard or soft error occurs.



### *sqlRecord*

Name of an SQL record that is used in a previously run **open** statement. You must specify either an SQL record or a result set ID, and it is recommended that you specify the result set ID.

### from *resultSetID*

The result-set identifier that is used in a previously run **open** statement. For details, see *resultSetID*.

### into ... *item*

An INTO clause, which identifies the EGL host variables that receive values

from the cursor or stored procedure. In a clause like this one (which is outside of a **#sql{ }** block), do not include a semicolon before the name of a host variable.

A specification of an INTO clause in this context overrides any INTO clause identified in the related **open** statement.

*statement*

A statement in the EGL language

In most cases, the EGL runtime issues an implicit **close** statement occurs after the last iteration of the **forEach** statement. That implicit statement modifies the SQL system variables, for which reason you may want to save the values of SQL-related system variables in the body of the **forEach** statement.

The EGL runtime does not issue an implicit **close** statement if the **forEach** statement ends because of an error other than **noRecordFound**.

An example is as follows, as further described in *SQL examples*:

```
VGVar.handleHardIOErrors = 1;

try
  open selectEmp
  with #sql{
    select empnum, empname
    from employee
    where empnum >= :empnum
    for update of empname
  }
  into empnum, empname;
onException
  myErrorHandler(6); // exits program
end

try
  forEach (from selectEmp)
    empname = empname + " " + "III";

    try
      execute
        #sql{
          update employee
          set empname = :empname
          where current of selectEmp
        };
    onException
      myErrorHandler(10); // exits program
    end
  end // end forEach; cursor is closed automatically
    // when the last row in the result set is read

onException
  // the exception block related to forEach is not run if the condition
  // is "sqlcode = 100", so avoid the test "if (sqlcode != 100)"
  myErrorHandler(8); // exits program
end

sysLib.commit();
```

**Related concepts**

“resultSetID” on page 867

“SQL support” on page 277



### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

### Related reference

"EGL statements" on page 88

"exit" on page 681

"open" on page 722

"SQL examples" on page 288

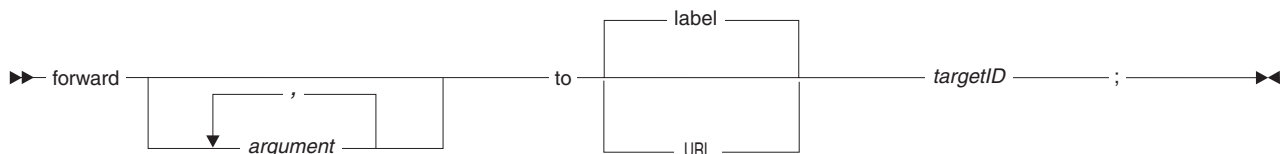
## forward

The EGL **forward** statement is used primarily to display a Web page with variable information, but can access a URL or can invoke a servlet or Java program that runs in the Web application server.

The statement acts as follows:

1. Commits recoverable resources, closes files, and releases locks
2. Forwards control
3. Ends the code that runs the **forward** statement

The syntax diagram is as follows:



### *argument*

An item or record that is passed to the code being invoked. The names of an argument and its corresponding parameter must be the same in all cases. You may not pass literals.

If you are invoking a PageHandler, the arguments must be compatible with the parameters specified for the **onPageLoad** function of the PageHandler. The function (if any) may have any valid name and is referenced by the PageHandler property **OnPageLoadFunction**. If you are invoking a program, the arguments must be compatible with the program parameters.

The following details may be of interest, depending on how you are using the technology:

- The argument must be named the same as the corresponding parameter because the name is used as a key in storing and retrieving the argument value on the Web application server.
- Instead of passing an argument, the invoker can do as follows before invoking the **forward** statement:
  - Place a value in the request block by invoking the system function `J2EELib.setRequestAttr`; or
  - Place a value in the session block by invoking the system function `J2EELib.setSessionAttr`.

In this case, the receiver does not receive the value as an argument, but by invoking the appropriate system function:

- `J2EELib.getRequestAttr` (to access data from the request block); or
- `J2EELib.getSessionAttr` (to access data from the session block).

- A character item is passed as an object of type Java String.
- A record is passed as a Java Bean.

**to label** *targetID*

Specifies a Java Server Faces (JSF) label, which identifies a mapping in a runtime JSF-based configuration file. The mapping in turn identifies the object to invoke, whether a JSP (usually one associated with an EGL PageHandler), an EGL program, a non-EGL program, or a servlet. The word **label** is optional, and *targetID* is a quoted string.

**to URL***targetID*

Specifies a URL and (if needed) a string of values used by the invoked PageHandler, VGWebTransaction program, or servlet.

**Related tasks**

“Forwarding control between pageHandlers and Web transactions” on page 165

**Related reference**

“Function invocations” on page 613

“getRequestAttr()” on page 932

“getSessionAttr()” on page 933

“transferName” on page 1076

## freeSQL

The EGL **freeSQL** statement frees any resources associated with a dynamically prepared SQL statement, closing any open cursor associated with that SQL statement.

►►—freeSQL *preparedStatementID*—————; —►►

*preparedStatementID*

An identifier that identifies a **prepare** statement. No error occurs if that statement did not run previously.

After you issue a **freeSQL** statement, you cannot run the **execute**, **open**, or **get** statement for the prepared SQL statement without reissuing the **prepare** statement.

**Related concepts**

“SQL support” on page 277

**Related reference**

“execute” on page 677

“get”

“open” on page 722

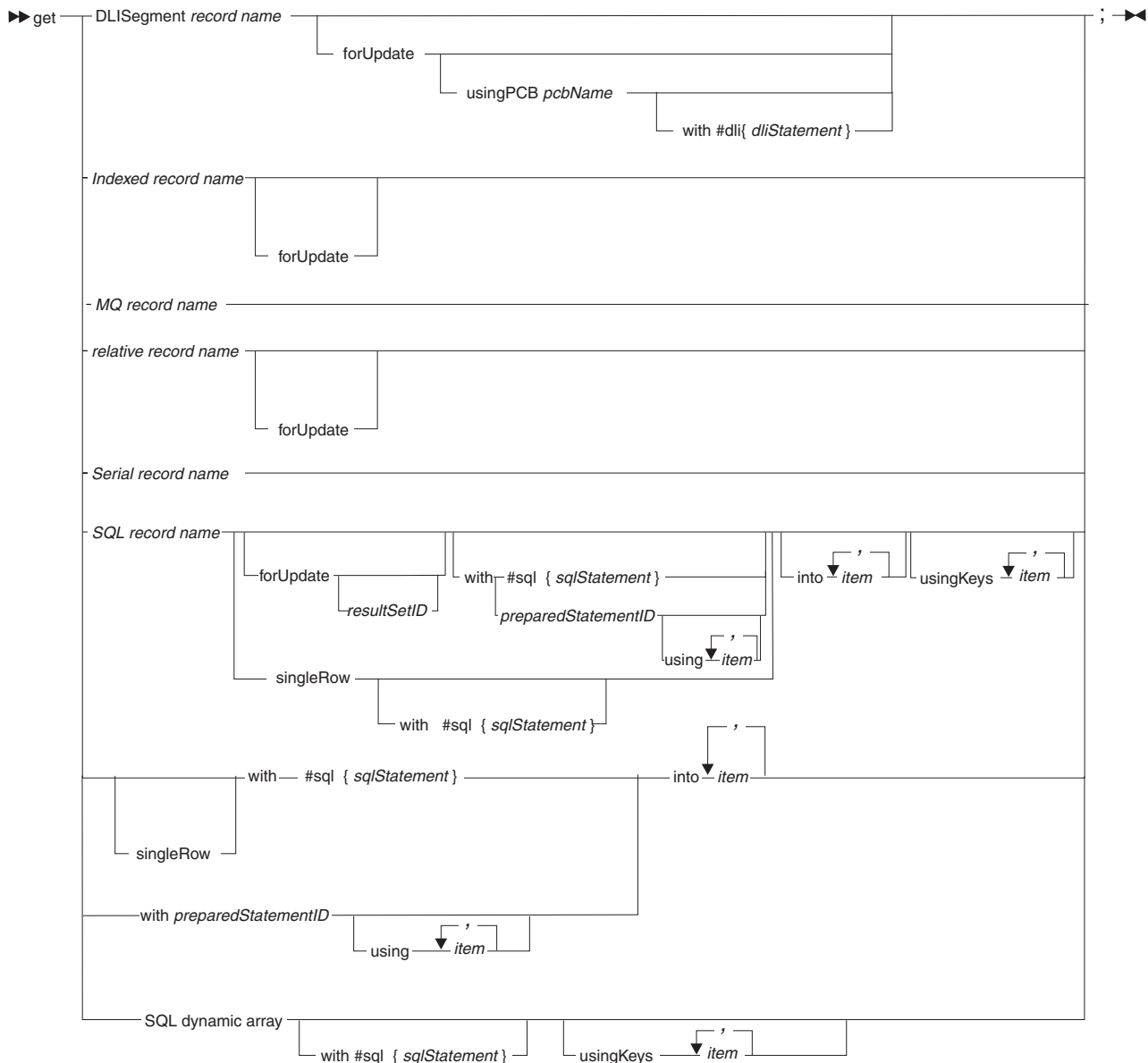
“prepare” on page 736

“Syntax diagram for EGL statements and commands” on page 884

## get

The EGL **get** statement retrieves a single file record or database row and provides an option that lets you replace or delete the stored data later in your code. In addition, this statement allows you to retrieve a set of database rows and place each succeeding row into the next SQL record in a dynamic array.

The **get** statement is sometimes identified as **get by key value** and is distinct from other statements that begin with the word *get*.



#### *record name*

Name of the I/O object to read: a DLISegment, indexed, MQ, relative, serial, or SQL record. For SQL processing, the record name is required if the EGL INTO clause (described later) is not specified.

#### **forUpdate**

Option that lets you use a later EGL statement to replace or delete the data that was retrieved from the file or database.

If the resource is recoverable (as in the case of a VSAM file or DL/I or SQL database), the **forUpdate** option locks the record so that it cannot be changed by other programs until a commit occurs. For details on commit processing, see *Logical unit of work*.

#### **usingPCB** *pcbName*

Option allowing you to specify the name of a PCB, as defined in your PSB record, to use instead of the default PCB.

**with #dli{ *dliStatement* }**

Option that allows an explicit DL/I GU or GHU statement, as described in *#dli directive*. Leave no space between #dli and the left brace.

*resultSetID*

A result-set identifier for use in an EGL **replace**, **delete**, or **execute** statement, as well as in an EGL **close** statement. For details, see *resultSetID*.

**singleRow**

Option that causes generation of more efficient SQL, as is appropriate when you are sure that the search criterion in the **get** statement applies to only one row and when you do not intend to update or delete the row. A runtime I/O error results if you specify this option when the search criterion applies to multiple rows. For additional details, see *SQL record*.

**#sql{ *sqlStatement* }**

Option that allows an explicit SQL SELECT statement, as described in *SQL support*. Leave no space between #sql and the left brace.

**into ... *item***

An EGL INTO clause, which identifies the EGL host variables that receive values from a relational database. This clause is required when you are processing SQL, in either of these cases:

- An SQL record is not specified; or
- Both an SQL record and an explicit SQL SELECT statement are specified, but a column in the SQL SELECT clause is not associated with a record item. (The association is in the SQL record part, as noted in *SQL item properties*.)

In a clause like this one (which is outside of an **#sql{ }** block), do not include a semicolon before the name of a host variable.

*preparedStatementID*

The identifier of an EGL **prepare** statement that prepares an SQL SELECT statement at run time. The **get** statement runs the SQL SELECT statement dynamically. For details, see *prepare*.

**using ... *item***

A USING clause, which identifies the EGL host variables that are made available to the prepared SQL SELECT statement at run time. In a clause like this one (which is outside of an **sql-and-end** block), do not include a semicolon before the name of a host variable.

**usingKeys ... *item***

Identifies a list of key items that are used to build the key-value component of the WHERE clause in an implicit SQL statement. The implicit SQL statement is used at run time if you do not specify an explicit SQL statement.

If you do not specify a **usingKeys** clause, the key-value component of the implicit statement is based on the SQL record part that is either referenced in the **get** statement or is the basis of the dynamic array referenced in the **get** statement.

In the case of a dynamic array, the items in the **usingKeys** clause (or the host variables in the SQL record) must *not* be in the SQL record that is the basis of the dynamic array.

The **usingKeys** information is ignored if you specify an explicit SQL statement.

*SQL dynamic array*

Name of a dynamic array that is composed of SQL records.

The following example shows how to read and replace a file record:

```
emp.empnum = 1;           // sets the key in record emp

try
  get emp forUpdate;
onException
  myErrorHandler(8); // exits the program
end

emp.empname = emp.empname + " Smith";

try
  replace emp;
onException
  myErrorHandler(12);
end
```

The next **get** statement uses the SQL record emp when retrieving a database row, with no subsequent update or deletion possible:

```
try
  get emp singleRow into empname with
    #sql{
      select empname
      from Employee
      where empnum = :empnum
    };
onException
  myErrorHandler(8);
end
```

The next example uses the same SQL record to replace an SQL row:

```
try
  get emp forUpdate into empname with
    #sql{
      select empname
      from Employee
      where empnum = :empnum
    };

onException
  myErrorHandler(8); // exits the program
end

emp.empname = emp.empname + " Smith";

try
  replace emp;
onException
  myErrorHandler(12);
end
```

In the CICS for MVS/ESA environment, the **get** position is lost when a commit or rollback is issued, or following a **converse** statement if running in segmented mode.

Details on the **get** statement depend on the record type. For details on DL/I processing, see *DLISegment record*. For details on SQL processing, see *SQL record*.

### **DLISegment record**

The **get** statement generates a DL/I GU statement. The **get...forUpdate** statement generates a DL/I GHU statement. The principal uses of the **get** statement are:

- To read a segment in order to do something with the data from that segment, such as print a report or an invoice.
- To hold a segment (in combination with the **forUpdate** option) in order to use the EGL keywords **delete** or **replace** to remove or update a segment. In most cases you do not need to perform a **get** statement before you **add** a record.

For an example of using the **get...forUpdate** statement with DL/I, see *delete*.

DL/I also supports the use of path calls on **get** statements. This means that you can read parent segments for all segment levels between the lowest level segment you are reading and the root. In the following example, DL/I will read all three segments (customer, location, and order) into their respective DLISegment records with a single call:

```
get myCustomer, myLocation, myOrder;
```

EGL will generate the following pseudo-DL/I code from this statement:

```
GU STSCCST*D (STQCCNO = :myCust.customerNo)
  STSCLOC (STQCLNO = :myLocation.locationNo)
  STPCORD
```

## Indexed record

When you issue a **get** statement against an indexed record, the key value in the record determines what record is retrieved from the file.

If you want to replace or delete an indexed (or relative) record, you must issue a **get** statement for the record and then issue the file-changing statement (**replace** or **delete**), with no intervening I/O operation against the same file. After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** statement on a record in the same file and includes the **forUpdate** option, a subsequent **replace** or **delete** statement is valid on the newly read file record
- If the next I/O operation is a **get** statement on the same EGL record (with no **forUpdate** option) or is a **close** statement on the same file, the file record is released without change

If the file is a VSAM file, the EGL **get** statement (with the **forUpdate** option) prevents the record from being changed by other programs. In z/OS batch programs, the lock remains until a commit occurs, which may not happen until the job step ends. In iSeries COBOL programs, the lock remains until a commit occurs, which may not happen until the end of the run unit, as described in *Run unit*.

## MQ record

When you read an MQ record in a message queue with the **get** keyword, EGL automatically:

1. Connects to the queue manager, if not already connected
2. Opens the queue, if the queue is not already open

3. Gets the next message from the queue and moves the message contents to the message queue record structure. The IMS message header (length, ZZ control information field, and transaction code) is automatically removed from each record read from the queue.

On IMS, a **get** statement for a serial record assigned to a single-segment message queue results in a get unique (GU) call to the I/O PCB. This GU call results in an automatic commit point. The first **get** statement for a serial record assigned to a multiple-segment message queue results in a GU call to the I/O PCB. Subsequent **get** statements result in GN calls until an NRF (no records found at this segment level) condition is reached. The first **get** statement after the NRF results in another GU call, and the process continues until an EOF is reached. Each GU call results in an automatic commit point.

### Relative record

When you issue a **get** statement against a relative record, the key item associated with the record determines what record is retrieved from the file. The key item must be available to any function that uses the record and can be any of these:

- An item in the same record
- An item in a record that is global to the program or is local to the function that is running the **get** statement
- A data item that is global to the program or is local to the function that is running the **get** statement

If you want to replace or delete an indexed (or relative) record, you must issue a **get** statement for the record and then issue the file-changing statement (**replace** or **delete**), with no intervening I/O operation against the same file. After you issue the **get** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** on the same file (with the forUpdate option), a subsequent replace or delete is valid on the newly read file record
- If the next I/O operation is a **get** on the same EGL record (with no forUpdate option) or is a close on the same file, the file record is released without change

### Serial records

A **get** statement reads the record following the last record read in the entry sequence. The first record is read for the first scan of a file. If the record accessed on the previous I/O operation was the last record in the file, the **get** statement returns EOF.

The file is closed and reopened whenever the program changes from adding to reading or from reading to adding. When the file is closed, the file position is lost; therefore, the first **get** statement after an **add** statement will read the first record from the file. Similarly, an **add** that follows a **get** or **get next** statement will add a record to the beginning of the file.

In the zSeries batch environment, a **get** statement for a serial record assigned to a GSAM file results in a get next (GN) call to the GSAM database. If a variable-length serial record is in a file associated with GSAM and the record

length is longer than the physical file, DL/I returns a blank status code. Data is truncated, but no message is issued because the situation cannot be detected.

### SQL record

The EGL **get** statement results in an SQL SELECT statement in the generated code. If you specify the `singleRow` option, the SQL SELECT statement is a stand-alone statement. Alternatively, the SQL SELECT statement is a clause in a cursor, as described in *SQL support*.

**Error conditions:** The following conditions are among those that are not valid when you use a **get** statement to read data from a relational database:

- You specify an SQL statement of a type other than SELECT
- You specify an SQL INTO clause directly in an SQL SELECT statement
- Aside from an SQL INTO clause, you specify some but not all of the clauses of an SQL SELECT statement
- You specify (or accept) an SQL SELECT statement that is associated with a column that either does not exist or is incompatible with the related host variable

The following error conditions are among those that can occur when you use the `forUpdate` option:

- You specify (or accept) an SQL statement that shows an intent to update multiple tables; or
- You use an SQL record as an I/O object, and all the record items are read only.

Also, the following situation causes an error:

- You customize an EGL **get** statement with the `forUpdate` option, but fail to indicate that a particular SQL table column is available for update; and
- The replace statement that is related to that **get** statement tries to revise the column.

You can solve the previous mismatch in any of these ways:

- When you customize the EGL **get** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL replace statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **get** and **replace** statements.

**Implicit SQL SELECT statement:** When you specify an SQL record as an I/O object for the **get** statement but do not specify an explicit SQL statement, the implicit SQL SELECT has the following characteristics:

- The record-specific property called **defaultSelectCondition** determines what table row is selected, so long as the value in each SQL table key column is equal to the value in the corresponding key item of the SQL record. If you specify neither a record key nor a default selection condition, all table rows are selected. If multiple table rows are selected for any reason, the first retrieved row is placed in the record.
- As a result of the association of record items and SQL table columns in the record definition, a given item receives the content of the related SQL table column.
- If you specify the `forUpdate` option, the SQL SELECT FOR UPDATE statement does not include record items that are read only.



- The SQL SELECT statement for a particular record is similar to the following statement, except that the FOR UPDATE OF clause is present only if the **get** statement includes the forUpdate option :

```
SELECT column01,
       column02, ...
       columnNN
FROM   tableName
WHERE  keyColumn01 = :keyItem01
FOR UPDATE OF
       column01,
       column02, ...
       columnNN
```

The SQL INTO clause on the standalone SQL SELECT or on the cursor-related FETCH statement is similar to this clause:

```
INTO   :recordItem01,
       :recordItem02, ...
       :recordItemNN
```

EGL derives the SQL INTO clause if the SQL record is accompanied by an explicit SQL SELECT statement when you have not specified an INTO clause. The items in the derived INTO clause are those that are associated with the columns listed in the SELECT clause of the SQL statement. (The item-and-column association is in the SQL record part, as noted in *SQL item properties*.) An EGL INTO clause is required if a column is not associated with an item.

When you specify a dynamic array of SQL records as an I/O object for the **get** statement but do not specify an explicit SQL statement, the implicit SQL SELECT is similar to that described for a single SQL record, with these differences:

- The key-value component of the query is a set of relationships that is based on a greater-than-or-equal-to condition:

```
keyColumn01 >= :keyItem01 &
keyColumn02 >= :keyItem02 &
.
.
.
keyColumnN  >= :keyItemN
```

- The items in the **usingKeys** clause (or the host variables in the SQL record) must *not* be in the SQL record that is the basis of the dynamic array.

### Related concepts

“DL/I database support” on page 310  
 “Logical unit of work” on page 395  
 “Record types and properties” on page 138  
 “References to parts” on page 23  
 “resultSetID” on page 867  
 “SQL support” on page 277

### Related tasks

“Syntax diagram for EGL statements and commands” on page 884

### Related reference

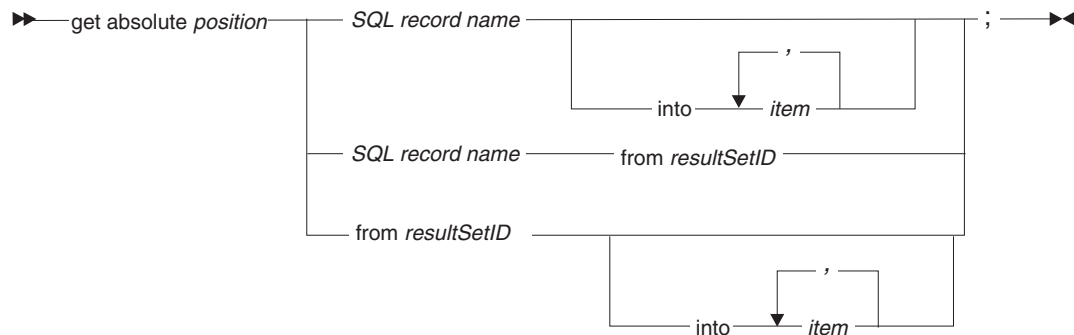
“#dli directive” on page 325  
 “add” on page 661  
 “close” on page 669  
 “CICS-related considerations” on page 530  
 “delete” on page 673  
 “EGL statements” on page 88

"Exception handling" on page 94  
 "execute" on page 677  
 "get next" on page 701  
 "get" on page 687  
 "get previous" on page 708  
 "I/O error values" on page 638  
 "open" on page 722  
 "prepare" on page 736  
 "replace" on page 738  
 "set" on page 742  
 "SQL item properties" on page 68  
 "terminalID" on page 1075

## get absolute

The EGL **get absolute** statement reads a numerically specified row in a relational-database result set. The row is identified in relation either to the beginning of the result set (if you specify a positive value) or to the end of the result set (if you specify a negative value).

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



### *position*

An integer item or literal.

If the value of *position* is positive, the row is identified in relation to the beginning of the result set. Specifying **get absolute 1**, for example, retrieves the first row and is equivalent to specifying **get first**. Specifying **get absolute 2** retrieves the second row.

If the value of *position* is negative, the row is identified in relation to the end of the result set. Specifying **get absolute -1**, for example, retrieves the last row and is equivalent to specifying **get last**. Specifying **get absolute -2** retrieves the second to last row.

A value of zero for *position* causes a hard error, as described in *Exception handling*.

### *record name*

Name of an SQL record.

**from *resultSetID***

An ID that ties the **get absolute** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

**into**

Begins an EGL into clause, which lists the items that receive values from a relational-database table.

***item***

An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get absolute** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get absolute** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If you issue a **get absolute** statement that attempts to access a row that is not in the result set, the EGL runtime acts as follows:

- Does not copy data from the result set
- Leaves the cursor open, with the cursor position unchanged
- Sets the SQL record (if any) to **noRecordFound**

In general, if an error occurs and processing continues, the cursor remains open, with the cursor position unchanged.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

**Related concepts**

"resultSetID" on page 867

"SQL support" on page 277

**Related tasks**

"Syntax diagram for EGL statements and commands" on page 884

**Related reference**

"delete" on page 673

"Exception handling" on page 94

"execute" on page 677

"get" on page 687

"get current" on page 697

"get first" on page 698

"get last" on page 699

"get next" on page 701

"get previous" on page 708

"get relative" on page 713

“EGL statements” on page 88

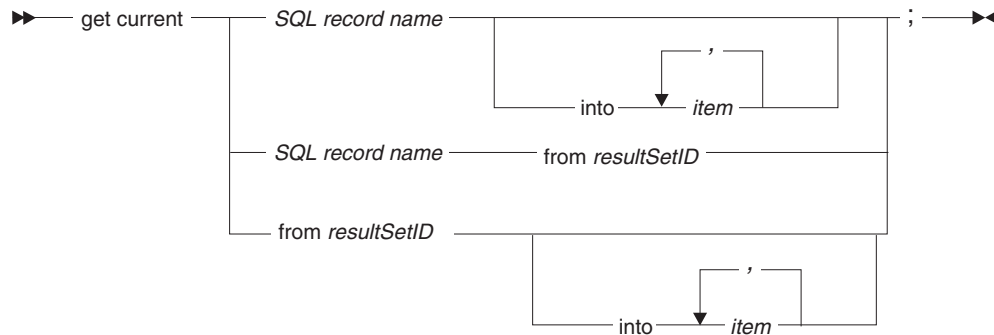
“open” on page 722

“replace” on page 738

## get current

The EGL **get current** statement reads the row at which the cursor is already positioned in a relational-database result set.

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



*record name*

Name of an SQL record.

**from** *resultSetID*

An ID that ties the **get current** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

**into**

Begins an EGL into clause, which lists the items that receive values from a relational-database table.

*item*

An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get current** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get current** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If an error occurs and processing continues, the cursor remains open.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

#### Related concepts

"resultSetID" on page 867

"SQL support" on page 277

#### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

#### Related reference

"delete" on page 673

"execute" on page 677

"get" on page 687

"get absolute" on page 695

"get first"

"get last" on page 699

"get next" on page 701

"get previous" on page 708

"get relative" on page 713

"EGL statements" on page 88

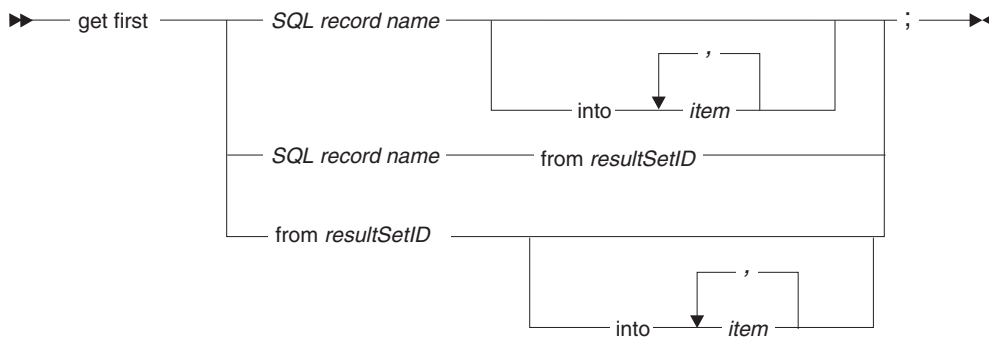
"open" on page 722

"replace" on page 738

## get first

The EGL **get first** statement reads the first row in a relational-database result set.

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



*record name*

Name of an SQL record.

**from** *resultSetID*

An ID that ties the **get first** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

**into**

Begins an EGL into clause, which lists the items that receive values from a relational-database table.

*item*

An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get first** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get first** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If an error occurs and processing continues, the cursor remains open.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

**Related concepts**

"resultSetID" on page 867

"SQL support" on page 277

**Related tasks**

"Syntax diagram for EGL statements and commands" on page 884

**Related reference**

"delete" on page 673

"execute" on page 677

"get" on page 687

"get absolute" on page 695

"get current" on page 697

"get last"

"get next" on page 701

"get previous" on page 708

"get relative" on page 713

"EGL statements" on page 88

"open" on page 722

"replace" on page 738

**get last**

The EGL **get last** statement reads the last row in a relational-database result set.

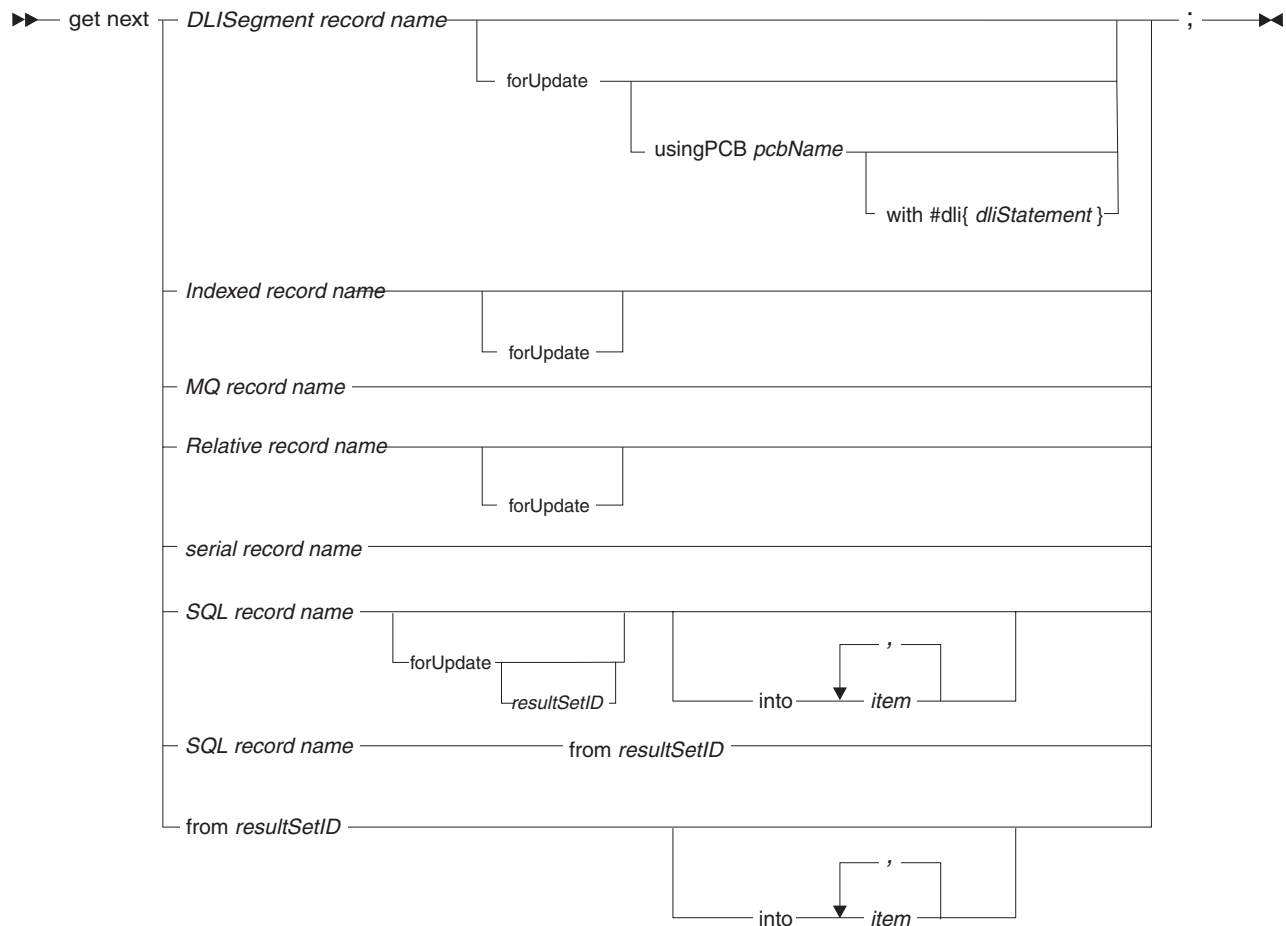
You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



“get absolute” on page 695  
 “get current” on page 697  
 “get first” on page 698  
 “get next”  
 “get previous” on page 708  
 “get relative” on page 713  
 “EGL statements” on page 88  
 “open” on page 722  
 “replace” on page 738

## get next

The EGL **get next** statement reads the next record from a file or message queue, or the next row from a database.



*record name*

Name of the I/O object: a DLISegment, indexed, MQ, relative, serial, or SQL record.

### forUpdate

Option that lets you use a later EGL statement to replace or delete the data that was retrieved from the file or database.

If the resource is recoverable (as in the case of an SQL database), the **forUpdate** option locks the record so that it cannot be changed by other programs until a commit occurs. For details on commit processing, see *Logical unit of work*.



**usingPCB** *pcbName*

Option allowing you to specify the name of a PCB, as defined in your PSB record, to use instead of the default PCB.

**with #dli{** *dliStatement* }

Option that allows an explicit DL/I GN or GHN statement, as described in *#dli directive*. Leave no space between #dli and the left brace.

**from** *resultSetID*

For SQL processing only, an ID that ties the **get next** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

**into**

Begins an EGL into clause, which lists the items that receive values from a relational-database table.

*item*

An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

An example of file access is as follows:

```

try
  open record1 forUpdate;
  onException
    myErrorHandler(8);
  return;
end
try
  get next record1;
  onException
    myErrorHandler(12);
  return;
end

while (record1 not endOfFile)
  makeChanges(record1); // process the record

  try
    replace record1;
    onException
      myErrorHandler(16);
    return;
  end

  try
    get next record1;
    onException
      myErrorHandler(12);
    return;
  end
end // end while

sysLib.commit();

```

Details on the **get next** statement depends on the record type. For details on SQL processing, see *SQL processing*.

**DLISegment record**

The **get next** statement generates a DL/I GN statement. The **get next...forUpdate** statement generates a DL/I GHN statement. The principal uses of the **get next** statement are as follows:

- To read a segment in order to do something with the data from that segment, such as print a report or an invoice.

- To hold a segment (in combination with the **forUpdate** option) in order to use the EGL keywords **delete** or **replace** to remove or update a segment. You do not need to perform a **get next** before you **add** a record, as the record will be added either in sequence or according the value of the key item for that record.

## Indexed record

When a **get next** statement operates on an indexed record, the effect is based on the current file position, which is set by either of these operations:

- A successful input or output (I/O) operation such as a **get** statement or another **get next** statement; or
- A **set** statement of the form *set record position*.

Rules are as follows:

- When the file is not open, the **get next** statement reads a record with the lowest key value in the file.
- Each subsequent **get next** statement reads a record that has the next highest key value in relation to the current file position. An exception for duplicate keys is described later.
- After a **get next** statement reads the record with the highest key value in the file, the next **get next** statement results in the I/O error value **endOfFile**.
- The current file position is affected by any of these operations:
  - An EGL **set** statement of the form *set record position* establishes a file position based on the *set value*, which is the key value in the indexed record that is referenced by the **set** statement. The subsequent **get next** statement against the same indexed record reads the file record that has a key value equal to or greater than the set value. If no such record exists, the result of the **get next** is **endOfFile**.
  - A successful I/O statement other than a **get next** statement establishes a new file position, and the subsequent **get next** statement issued against the same EGL record reads the *next* file record. After a **get previous** statement reads a file record, for example, the **get next** statement either reads the file record with the next-highest key or returns **endOfFile**.
  - If a **get previous** statement returns **endOfFile**, the subsequent **get next** statement retrieves the first record in the file.
  - After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** or **get previous** statement.
- When you are using an alternate index and duplicate keys are in the file, the following rules apply:
  - Retrieval of a record with a higher-valued key occurs only after **get next** statements have read all the records that have the same key as the most recently retrieved record. The order in which duplicate-keyed records are retrieved is the order in which VSAM returns the records.
  - If a **get next** follows a successful I/O operation other than a **get next**, the **get next** skips over any duplicate-keyed records and retrieves the record with the next higher key.
  - The EGL error value **duplicate** is not set when your program retrieves the last record in a group of records containing the same key.

Consider a file in which the keys are as follows:

1, 2, 2, 2, 3, 4

Each of the following tables illustrates the effect of running a sequence of EGL statements on the same indexed record.

The next two tables apply to EGL-generated COBOL code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
<b>get</b>	2	2 (the first of three)	duplicate
<b>get next</b>	any	3	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
<b>set</b> (of the form <i>set record position</i> )	2	no retrieval	duplicate
<b>get next</b>	any	2 (the first of three)	duplicate
<b>get next</b>	any	2 (the second)	duplicate
<b>get next</b>	any	2 (the third)	—
<b>get next</b>	any	3	—

The next two tables apply to EGL-generated Java code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
<b>get</b>	2	2 (the first of three)	duplicate
<b>get next</b>	any	2 (the second)	duplicate
<b>get next</b>	any	2 (the third)	—
<b>get next</b>	any	3	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
<b>set</b> (of the form <i>set record position</i> )	2	no retrieval	duplicate
<b>get next</b>	any	2 (the first of three)	—
<b>get next</b>	any	2 (the second)	duplicate
<b>get next</b>	any	2 (the third)	—
<b>get next</b>	any	3	—

## Message queue

When a **get next** operates on a MQ record, the first record in the queue is read into the MQ record. This placement occurs because the **get next** invokes one or more MQSeries calls:

- MQCONN connects the generated code to the default queue manager and is invoked when no connection is active
- MQOPEN establishes a connection to the queue and is invoked when a connection is active but the queue is not open

- MQGET removes the record from the queue and is always invoked unless an error occurred in an earlier MQSeries call

### Relative record

When a **get next** statement operates on a relative record, the effect is based on the current file position, which is set by a successful input or output (I/O) operation such as a **get** statement or another **get next** statement. Rules are as follows:

- When the file is not open, the **get next** statement reads the first record in the file.
- Each subsequent **get next** reads a record that has the next highest key value in relation to the current file position.
- A **get next** does not return **noRecordFound** if the next record is deleted. Instead, the **get next** skips deleted records and retrieves the next record in the file.
- After a **get next** statement reads the record with the highest key value in the file, the next **get next** statement results in the EGL error value **endOfFile**.
- The current file position is affected by any of these operations:
  - A successful I/O statement other than a **get next** establishes a new file position, and the subsequent **get next** against the same EGL record reads the *next* file record.
  - After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** statement.
- After a **get next** statement reads the last record in the file, the next **get next** statement results in the EGL error values **endOfFile** and **noRecordFound**.

### Serial record

When a **get next** statement operates on a serial record, the effect is based on the current file position, which is set by another **get next** statement. Rules are as follows:

- When the file is not open, the **get next** statement reads the first record in the file.
- Each subsequent **get next** statement reads the next record.
- After a **get next** statement reads the last record, the subsequent **get next** statement results in the EGL error value **endOfFile**.
- If the generated code adds a serial record and then issues the equivalent of a **get next** statement on the same file, EGL closes and reopens the file before executing the **get next** statement. A **get next** statement that follows an **add** statement therefore reads the first record in the file. This behavior also occurs when the **get next** and **add** statements are in different programs, and one program calls another. Similarly, an **add** that follows a **get** or **get next** statement will add a record to the beginning of the file.

It is recommended that you avoid having the same file open in more than one program at the same time.

On CICS for z/OS, a single program cannot include the combination of **add** statement and **get next** statement for the same spool file. This restriction also applies when the **get next** and **add** statements are in different programs, and one program calls the other.

On IMS/VS, the following considerations apply:

- A serial record must be associated with the I/O PCB (PCB 0). The **get next** statement is not supported for a transaction program or for a batch program that is called from a transaction program. Batch programs can use only one serial file

for input. The IMS message header (length, ZZ field, and transaction code) is automatically removed from each record read from the queue.

- A **get next** statement for a serial record assigned to a single-segment message queue results in a get unique (GU) call to the I/O PCB. This GU call results in an automatic commit point.
- The first **get next** statement for a serial record assigned to a multiple-segment message queue results in a GU call to the I/O PCB. Subsequent **get next** statements result in GN calls until an NRF (status code QD) condition is reached. The first **get next** statement after the NRF results in another GU call, and the function continues until an EOF (status code QC) is reached. Each GU call results in an automatic commit point.
- During any specific scheduling of a batch program, the program can do a **get next** statement from only one message queue. The transaction code for which IMS scheduled the program determines the message queue that is read. The system resource specified during generation is ignored.

For an IMS transaction-oriented BMP, a **get next** statement for a serial record assigned to a message queue involves the same considerations as in IMS/VS.

## SQL processing

When a **get next** statement operates on an SQL record, your code reads the next row from those selected by an **open** statement. If you issue a **get next** statement to retrieve a row that was selected by an **open** statement that has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get next** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If you issue a **get next** statement that attempts to access a row that is beyond the last selected row, the following statements apply:

- No data is copied from the result set
- EGL sets the SQL record (if any) to **noRecordFound**
- If the related **open** statement included the scroll option, the cursor remains open with the cursor position unchanged. The scroll option is valid only if you are generating output in Java.
- If you have not set the scroll option, the cursor is closed.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

## Related concepts

“DL/I database support” on page 310

“Record types and properties” on page 138

“resultSetID” on page 867

“SQL support” on page 277

## Related tasks

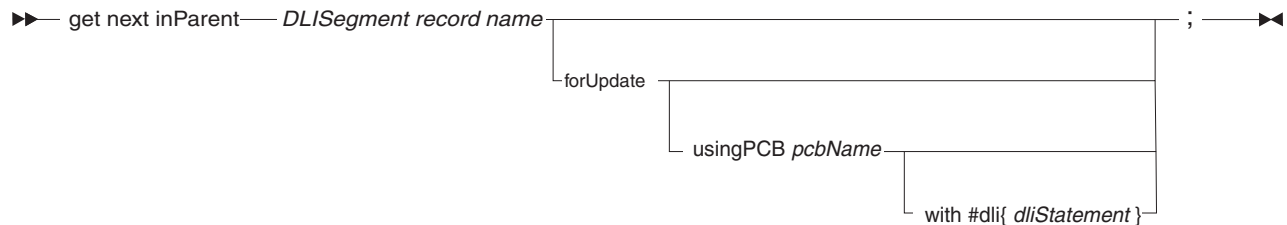
“Syntax diagram for EGL statements and commands” on page 884

### Related reference

"#dli directive" on page 325  
"add" on page 661  
"CICS-related considerations" on page 530  
"close" on page 669  
"delete" on page 673  
"Exception handling" on page 94  
"execute" on page 677  
"get" on page 687  
"get" on page 687  
"get previous" on page 708  
"I/O error values" on page 638  
"EGL statements" on page 88  
"open" on page 722  
"prepare" on page 736  
"replace" on page 738  
"set" on page 742

## get next inParent

As used for DL/I access, the EGL **get next inParent** statement reads the next child segment that has the same parent as the segment at the current database position.



### *record name*

Name of the I/O object which receives the values from the segment read. EGL currently supports this keyword for DLISegment records only.

### **forUpdate**

Option that lets you use a later EGL statement to replace or delete the data that was retrieved from the file or database.

If the resource is recoverable, the **forUpdate** option locks the record so that it cannot be changed by other programs until a commit occurs. For details on commit processing, see *Logical unit of work*.

### **usingPCB** *pcbName*

Option allowing you to specify the name of a PCB, as defined in your PSB record, to use instead of the default PCB.

### **with #dli{** *dliStatement* **}**

Option that allows an explicit DL/I GNP or GHNP statement, as described in *#dli directive*. Leave no space between **#dli** and the left brace.

In the following snippet, the code prints information for every order (child segment) in a location (parent segment) for a customer:

```
while (record1 not endOfFile)
  try
    get next inparent record1;
  onException
    myErrorHandler(12);
```

```

end
ordersForLocation = ordersForLocation + 1;
printOrderDetail(record1);
end // end while

```

## DL/I record

The **get next inParent** statement generates a DL/I GNP statement (no "forUpdate" modifier) or a GHNP (with "forUpdate" modifier). The principal uses of the **get next inParent** statement are as follows:

- To read a segment in order to do something with the data from that segment, such as print a report or an invoice.
- To hold a segment (in combination with the **forUpdate** option) in order to use the EGL keywords **delete** or **replace** to remove or update a segment. You do not need to perform a **get next inParent** before you **add** a record, as the record will be added either in sequence or according the value of the key item for that record.

## Related concepts

"DL/I database support" on page 310

"Record types and properties" on page 138

## Related tasks

"Syntax diagram for EGL statements and commands" on page 884

## Related reference

"#dli directive" on page 325

"add" on page 661

"CICS-related considerations" on page 530

"delete" on page 673

"Exception handling" on page 94

"get" on page 687

"get next" on page 701

"I/O error values" on page 638

"EGL statements" on page 88

"replace" on page 738

"set" on page 742

## get previous

The EGL **get previous** statement either reads the previous row from a relational-database result set or reads the previous record in the file that is associated with a specified EGL indexed record.

You can use this statement for a relational-database result set only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.





- A **set statement** of the form *set record position*.

Rules for an indexed record are as follows:

- When the file is not open, the **get previous** statement reads a record with the highest key value in the file.
- Each subsequent **get previous** reads a record that has the next lowest key value in relation to the current file position. An exception for duplicate keys is described later.
- After a **get previous** statement reads the record with the lowest key value in the file, the next **get previous** statement results in the EGL error value **endOfFile**.
- The current file position is affected by any of these operations:
  - An EGL **set statement** of the form *set record position* establishes a file position based on the *set value*, which is the key value in the indexed record that is referenced by the **set statement**. The subsequent **get previous** statement against the same indexed record reads the file record that has a key value equal to or less than the set value. If no such record exists, the result of the **get previous** statement is **endOfFile**.  
If the set value is filled with hexadecimal FF, the result of a **set** statement of the form *set record position* is as follows:
    - The **set** statement establishes a file position after the last record in the file
    - If a **get previous** statement is the next I/O operation, the generated code retrieves the last record in the file
  - A successful I/O statement other than a **get previous** statement establishes a new file position, and the subsequent **get previous** statement against the same EGL record reads the *previous* file record. After a **get next** statement reads a file record, for example, the **get previous** statement either reads the file record with the next-lowest key or returns **endOfFile**.
  - If a **get next** statement returns **endOfFile**, the subsequent **get previous** statement retrieves the last record in the file.
  - After an unsuccessful **get**, **get next**, or **get previous** statement, the file position is undefined and must be re-established by a **set** statement of the form *set record position* or by an I/O operation other than a **get next** or **get previous** statement.
- When you are using an alternate index and duplicate keys are in the file, the following rules apply:
  - Retrieval of a record with a lower-valued key occurs only after **get previous** statements have read all the records that have the same key as the most recently retrieved record. The order in which duplicate-keyed records are retrieved is the order in which VSAM returns the records.
  - If a **get previous** statement follows a successful I/O operation other than a **get previous**, the **get previous** statement skips over any duplicate-keyed records and retrieves the record with the next lower key.
  - The EGL error value **duplicate** is not set when your program retrieves the last record in a group of records containing the same key.

Consider a file in which the keys in an alternate index are as follows:

1, 2, 2, 2, 3, 4

Each of the following tables illustrates the effect of running a sequence of EGL statements on the same indexed record.

The next three tables apply to EGL-generated COBOL code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
get	3	3	—
get previous	any	2 (the first of three)	duplicate
get previous	any	2 (the second)	duplicate
get previous	any	2 (the third)	—
get previous	any	1	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
set (of the form <i>set record position</i> )	2	—	—
get next	any	2 (the first)	duplicate
get next	any	2 (the second)	—
get previous	any	1	—
get previous	any	--	endOfFile

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for COBOL
set (of the form <i>set record position</i> )	1	--	--
get previous	any	1	--

The next three tables apply to EGL-generated Java code.

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
get	3	3	—
get previous	any	2 (the first of three)	duplicate
get previous	any	2 (the second)	duplicate
get previous	any	2 (the third)	—
get previous	any	1	—

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
set (of the form <i>set record position</i> )	2	—	duplicate
get next	any	2 (the first)	—
get next	any	2 (the second)	duplicate
get previous	any	1	—
get previous	any	--	endOfFile

EGL statement (in order)	Key in the indexed record	Key in the file record retrieved by the statement	EGL error value for Java
<b>set</b> (of the form <i>set record position</i> )	1	--	--
<b>get previous</b>	any	1	--

## SQL processing

When a **get previous** statement operates on an SQL record, your code reads the previous row from those selected by an **open** statement, but only if you have specified the scroll option. If you issue a **get previous** statement to retrieve a row that was selected by an **open** statement that also has the forUpdate option, you can do any of these:

- Change the row with an EGL **replace** statement
- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get previous** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If you issue a **get previous** statement that attempts to access a row that is previous to the first selected row, the EGL runtime acts as follows:

- Does not copy data from the result set
- Leaves the cursor open, with the cursor position unchanged
- Sets the SQL record (if any) to **noRecordFound**

In general, if an error occurs and processing continues, the cursor remains open, with the cursor position unchanged.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

### Related concepts

"Record types and properties" on page 138

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

### Related reference

"add" on page 661

"close" on page 669

"delete" on page 673

"Exception handling" on page 94

"execute" on page 677

"get" on page 687

"get next" on page 701

"I/O error values" on page 638

"open" on page 722

"prepare" on page 736

"EGL statements" on page 88

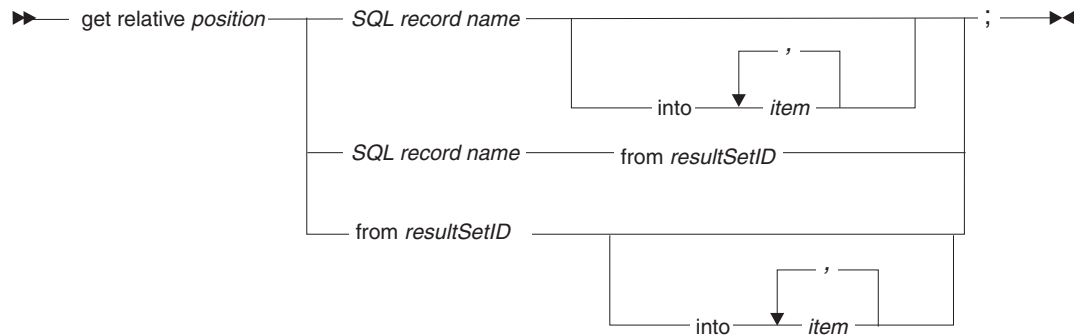
"replace" on page 738

"set" on page 742

## get relative

The EGL **get relative** statement reads a numerically specified row in a relational-database result set. The row is identified in relation to the cursor position in the result set.

You can use this statement only if you specified the scroll option in the related **open** statement. The scroll option is available only if you are generating output in Java.



### *position*

An integer item or literal.

If the value of *position* is positive, the position is an increment to the current numeric position in the result set. Specifying **get relative 2** when the cursor is on the first row, for example, retrieves the third row; and specifying **get relative 1** is equivalent to specifying **get next**.

If the value of *position* is negative, the position is a decrement to the current numeric position in the result set. Specifying **get relative -2** when the cursor is on the third row, for example, retrieves the first row; and specifying **get relative -1** is equivalent to specifying **get previous**.

A value of zero for *position* retrieves the row at the cursor position already in effect and is equivalent to specifying **get current**.

### *record name*

Name of an SQL record.

### **from** *resultSetID*

An ID that ties the **get relative** statement to an **open** statement run earlier in the same program. For details, see *resultSetID*.

### **into**

Begins an EGL into clause, which lists the items that receive values from a relational-database table.

### *item*

An item that receives the value of a particular column. Do *not* precede the item name with a colon (:).

If you issue a **get relative** statement to retrieve a row that was selected by an **open** statement that has the `forUpdate` option, you can do any of these:

- Change the row with an EGL **replace** statement

- Remove the row with an EGL **delete** statement
- Change or remove the row with an EGL **execute** statement

An SQL FETCH statement represents the EGL **get relative** statement in the generated code. The format of the generated SQL statement cannot be changed, except to set the INTO clause.

If you issue a **get relative** statement that attempts to access a row that is not in the result set, the EGL runtime acts as follows:

- Does not copy data from the result set
- Leaves the cursor open with the cursor position unchanged
- Sets the SQL record (if any) to **noRecordFound**

In general, if an error occurs and processing continues, the cursor remains open with the cursor position unchanged.

Finally, when you specify SQL COMMIT or sysLib.commit, your code retains position in the cursor that was declared in the **open** statement, but only if you use the hold option in the **open** statement.

#### Related concepts

"resultSetID" on page 867

"SQL support" on page 277

#### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

#### Related reference

"delete" on page 673

"Exception handling" on page 94

"execute" on page 677

"get" on page 687

"get absolute" on page 695

"get current" on page 697

"get first" on page 698

"get last" on page 699

"get next" on page 701

"get previous" on page 708

"EGL statements" on page 88

"open" on page 722

"replace" on page 738

## goTo

The EGL **goTo** statement causes processing to continue at a specified label, which must be in the same function as the statement and outside of a block.

► goto *label* : \_\_\_\_\_ ◀

*label*

A series of characters that are displayed elsewhere in the function, outside of any blocks, including these:

- if

- else
- when (in a **case** statement)
- while
- try

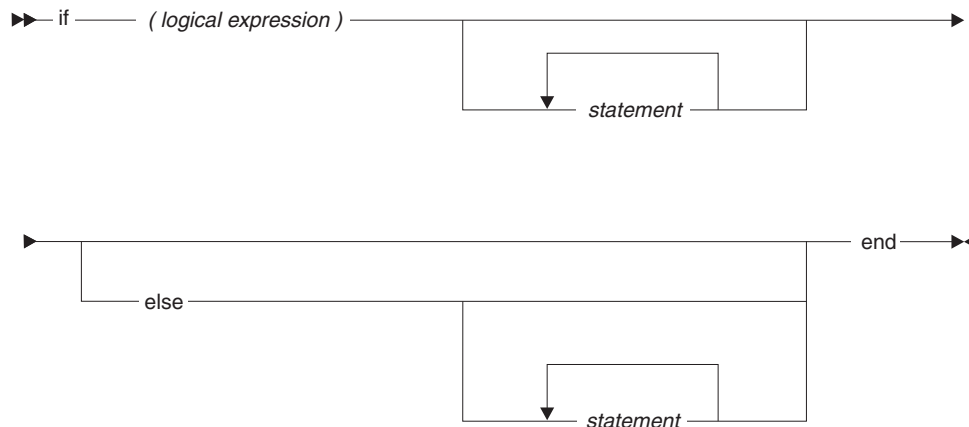
When displayed at the location where processing continues, the label is followed by colon. For details on valid characters for the label, see *Naming conventions*.

#### Related reference

"Naming conventions" on page 778

## if, else

The EGL keyword **if** marks the start of a set of statements (if any) that run only if a logical expression resolves to true. The optional keyword **else** marks the start of an alternative set of statements (if any) that run only if the logical expression resolves to false. The keyword **end** marks the close of the *if* statement.



#### *logical expression*

An expression (a series of operands and operators) that evaluates to true or false

#### *statement*

One or more EGL statements

You may nest **if** and other end-terminated statements to any level. Each **end** keyword refers to the most recent statement that was not ended and that begins with one of these keywords:

- **if**
- **case**
- **try**
- **while**

None of those statements is followed by a semicolon.

An example is as follows:

```

if (userRequest == "U")
  try
    update myRecord;
  onException

```

```

        myErrorHandler(12); // ends program
    end
    try
        myRecord.myItem=25;
        replace record1;
        onException
            myErrorHandler(16);
        end
    else
        try
            add record2;
            onException
                myErrorHandler(18); // ends program
            end
        if (sysVar.systemType is WIN)
            myFunction01();
        else
            myFunction02();
        end
    end
end

```

### Related tasks

“Syntax diagram for EGL statements and commands” on page 884

### Related reference

“Logical expressions” on page 593

“EGL statements” on page 88

## move

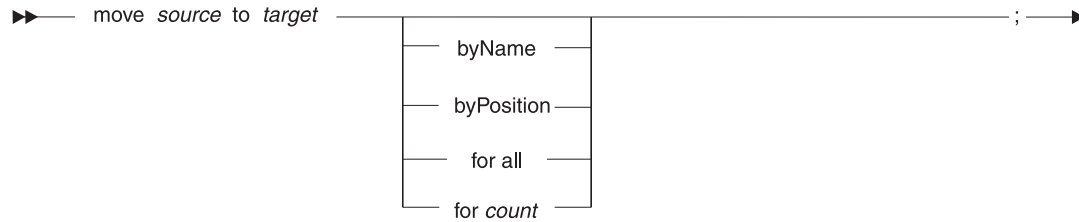
The EGL **move** statement copies data in any of three ways. The first option copies data byte by byte; the second (called *by name*) copies data from the named fields in one structure to the same-named fields in another; and the third (called *by position*) copies data from each field in one structure to the field at the equivalent position in another.

The following general rules apply:

- If the source value is one of these, the default is to copy data byte by byte--
  - A primitive variable
  - A field that is in a fixed structure
  - A literal
  - A constant

Otherwise, the default is to copy data by name.

- Moves are checked for field-to-field compatibility. The rules for truncation, padding, and type conversion are the same as those detailed for the **assignment** statement; however, the overall behavior of the **move** statement is different from that of the **assignment** statement.
- When you are working with dynamic arrays, the last element is determined by the array’s current size. The **move** statement never adds an element to an array; to expand a dynamic array, use the array-specific functions `appendElement` or `appendAll`, as described in *Arrays*.



The statement is best understood by reference to the following categories:

### **byName**

When you specify **byName**, data is written from each field in the source to a same-named field in the target. The operation occurs in the order in which the fields are in the source.

Source and target can be as follows:

#### *source*

One of these:

- A dynamic array of fixed records; but the array is valid only if the target is not a record
- A record
- A fixed record
- A structure field with a substructure
- A structure-field array with a substructure; but this array is valid only if the target is not a record
- A dataTable
- A form

A fixed-structure field whose name is an asterisk (\*) is not available as a source field, but any named fields in a substructure of that field are available.

#### *target*

One of these:

- A dynamic array of fixed records; but this array is valid only if the source is not a record
- A record
- A fixed record
- A structure field with a substructure
- A structure-field array with a substructure; but this array is valid only if the source is not a record
- A dataTable
- A form
- 

An example statement is as follows:

```
move myRecord01 to myRecord02 byName;
```

The operation is not valid in any of these cases:

- Two or more fields in the source have the same name;
- Two or more fields in the destination have the same name;



- The source field is either a multidimensional structure-field array or a one-dimensional structure-field array whose container is an array; or
- The target field is either a multidimensional structure-field array or a one-dimensional structure-field array whose container is an array.

The operation works as follows:

- In a simple case, the source is a fixed structure but is not itself an array element, and the same is true of the target. These rules apply--
  - If no arrays are involved, the value of each subordinate field in the source structure is copied to the same-named field in the target structure.
  - If an array of structure fields is being copied to an array of structure fields, the operation is treated as a *move for all*:
    - The elements of the source field are copied to successive elements of the target field
    - If the source array has fewer elements than the target array, processing stops when the last element of the source array is copied
- In another case, the source or target is a record. The fields of the source are assigned to the same-named fields in the target.
- A less simple case is best introduced by example. The source is an array of 10 fixed records, each of which includes these structure fields:

```
10 empnum CHAR(3);
10 empname CHAR(20);
```

The target is a fixed structure that includes these structure fields:

```
10 empnum CHAR(3)[10];
10 empname CHAR(20)[10];
```

The operation copies the value of field empnum in the first fixed record to the first element of the structure-field array empnum; copies the value of field empname in the first fixed record to the first element of the structure-field array empname; and does a similar operation for each fixed record in the source array.

The equivalent operation occurs if the source is a single fixed record that has a substructure like this:

```
10 mySubStructure[10]
15 empnum CHAR(3);
15 empname CHAR(20);
```

- Finally, consider the case in which the source is a fixed record that includes these structure fields:

```
10 empnum CHAR(3);
10 empname CHAR(20)[10];
```

The target is a form, fixed record, or structure field that has the following substructure:

```
10 empnum char(3)[10];
10 empname char(20);
```

The value of field empnum is copied from the source to the first element of empnum in the target; and the value of the first element of empname is copied from the source to the field empname in the target.

### byPosition

The purpose of **byPosition** is to copy data from each field in one structure to the field at the equivalent position in another.

Source and target can be as follows:

*source*

One of these:

- A dynamic array of fixed records; but the array is valid only if the target is not a record
- A record
- A fixed record
- A structure field with a substructure
- A structure-field array with a substructure; but this array is valid only if the target is not a record
- A dataTable

*target*

One of these:

- A dynamic array of fixed records; but this array is valid only if the source is not a record
- A record
- A fixed record
- A structure field with a substructure
- A structure-field array with a substructure; but this array is valid only if the source is not a record
- A dataTable

When you move data between a record and a fixed structure, only the top-level fields of the fixed structure are considered. When you move data between two fixed structures, only the lowest-level (leaf) fields of either structure are considered.

The operation is not valid if the source or target field is a multidimensional structure-field array, or a one-dimensional structure field array whose container is an array.

The operation works as follows:

- In a simple case, the source is a fixed structure but is not itself an array element, and the same is true of the target. These rules apply--
  - If no arrays are involved, the value of each leaf field in the source structure is copied to the leaf field in the target structure at the corresponding position.
  - If an array of structure fields is being copied to an array of structure fields, the operation is treated as a *move for all*:
    - The elements of the source field are copied to successive elements of the target field
    - If the source array has fewer elements than the target array, processing stops when the last element of the source array is copied
- In another case, the source or target is a record. The top-level or leaf fields of the source (depending on the source type) are assigned to the top-level or leaf fields in the target (depending on the target type).
- A less simple case is best introduced by example. The source is an array of 10 fixed records, each of which includes these structure fields:

```
10 empnum CHAR(3);  
10 empname CHAR(20);
```

The target is a fixed structure that includes these structure fields:

```
10 empnum CHAR(3)[10];
10 empname CHAR(20)[10];
```

The operation copies the value of field empnum in the first fixed record to the first element of the structure-field array empnum; copies the value of field empname in the first fixed record to the first element of the structure-field array empname; and does a similar operation for each fixed record in the source array.

The equivalent operation occurs if the source is a single fixed record that has a substructure like this:

```
10 mySubStructure[10]
15 empnum CHAR(3);
15 empname CHAR(20);
```

- Finally, consider the case in which the source is a fixed record that includes these structure fields:

```
10 empnum CHAR(3);
10 empname CHAR(20)[10];
```

The target is a form, fixed record, or structure field that has the following substructure:

```
10 empnum char(3)[10];
10 empname char(20);
```

The value of field empnum is copied from the source to the first element of empnum in the target; and the value of the first element of empname is copied from the source to the field empname in the target.

## for all

The purpose of **for all** is to assign values to all elements in a target array.

Source and target can be as follows:

### source

One of these:

- A dynamic array of records, fixed records, or primitive variables
- A record
- A fixed record
- A structure field with or without a substructure
- A structure-field array with or without a substructure
- A primitive variable
- A literal or constant

### target

One of these:

- A dynamic array of records, fixed records, or primitive variables
- A structure-field array with or without a substructure
- An element of a dynamic or structure-field array

The **move** statement in this case is equivalent to multiple **assignment** statements, one per target array element, and an error occurs if an attempted assignment is not valid. For details on validity, see *Assignments*.

If a source or target element has a fixed structure, the **move** statement treats that structure as a field of type CHAR unless the top level of the structure specifies a different primitive type. When **for all** is in use, the **move** statement gives no consideration to substructure.

If the source is an element of an array, the source is treated as an array in which the specified element is the first element, and previous elements are ignored.

If the source is an array or an element of an array, each successive element of the source array is copied to the sequentially next element of the target array. Either the target array or the source array can be longer, and the operation ends when data is copied from the last element having a matching element in the other array.

If the source is neither an array nor an element of an array, the operation uses the source value to initialize every element of the target array.

#### **for count**

The purpose of **for count** is to assign values to a sequential subset of elements in a target array. Examples are as follows:

- The next statement moves "abc" to elements 7, 8, and 9 in target:  
    move "abc" to target[7] for 3
- The next statement moves elements 2, 3, and 4 from source into elements 7, 8, and 9 in target:  
    move source[2] to target[7] for 3

The operation works as follows:

- If the source is neither an array nor an element of an array, the operation uses the source value to initialize elements of the target array.
- If the source is an array, the first element of that array is the first in a set of elements to be copied. If the source is an element of an array, that element is the first in a set of elements to be copied.
- If the target is an array, the first element of that array is the first in a set of elements to receive data. If the target is an element of an array, that element is the first in a set of elements that receives data.

The *count* value indicates how many target elements are to receive data. The value can be any of these:

- An integer literal
- A variable that resolves to an integer
- A numeric expression, but not a function invocation

The **move** statement is equivalent to multiple **assignment** statements, one per target array element, and an error occurs if an attempted assignment is not valid. For details on validity, see *Assignments*.

If a source or target element has an internal structure, the **move** statement treats that structure as a field of type CHAR unless the top level of that structure specifies a different primitive type. When **for count** is in use, the **move** statement gives no consideration to substructure.

When the source and target are both arrays, either the target array or the source array can be longer, and the operation ends after the first of two events occurs:

- Data is copied between the last elements for which the operation is requested; or
- Data is copied from the last element having a matching element in the other array.

When the source is not an array, the operation ends after the first of two events occurs:

- Data is copied to the last element for which the operation is requested; or
- Data is copied to the last element in the array.

If the source is a record array (or an element of one), the target must be a record array. If the source is a primitive-variable array (or an element of one), the target must be either a primitive-variable array or a structure-field array. If the source is a structure-field array (or an element of one), the target must be either a primitive-variable array or a structure-field array.

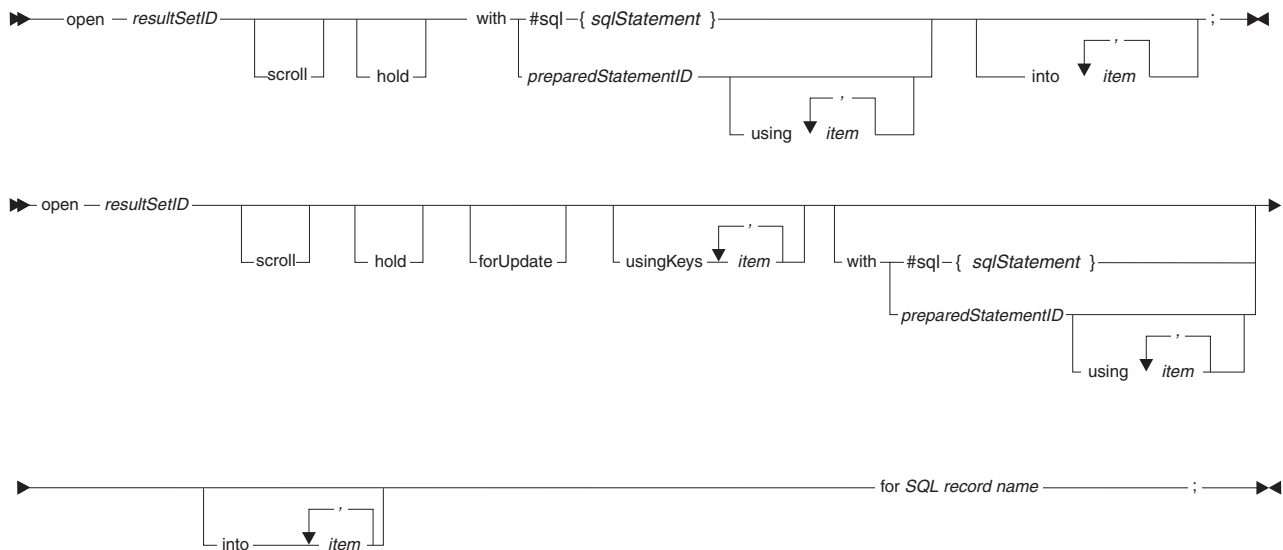
#### Related reference

“Arrays” on page 75

“Assignments” on page 456

## open

The EGL **open** statement selects a set of rows from a relational database for later retrieval with **get next** statements. The **open** statement may operate on a cursor or on a called procedure.



#### resultSetID

ID that ties the open statement to later **get next**, **replace**, **delete**, and **close** statements. For details, see *resultSetID*.

#### scroll

Option that lets you move through a result set in various ways. The statement **get next** is always available to you, but use of **scroll** allows you to use the following statements too:

- **get absolute**
- **get current**
- **get first**
- **get last**
- **get previous**
- **get relative**

The **scroll** option is available only if you are generating output in Java.

### **hold**

Maintains position in a result set when a commit occurs. The **hold** option is available for Java programs only if the JDBC driver supports JDBC 3.0 or higher. The option is available for COBOL programs; however, if a program targeted for CICS is segmented, the option **hold** has little value because a converse in a segmented program ends the CICS transaction and prevents the program from retaining any file or database position.

The **hold** option is appropriate in the following case:

- You are using the EGL **open** statement to open a cursor rather than a stored procedure; and
- You want to commit changes periodically without losing your position in the result set; and
- Your database management system supports use of the WITH HOLD option in the SQL cursor declaration.

Your code might do as follows, for example:

1. Declare and open a cursor by running an EGL **open** statement
2. Fetch a row by running an EGL **get next** statement
3. Do the following in a loop:
  - a. Process the data in some way
  - b. Update the row by running an EGL **replace** statement
  - c. Commit changes by running the system function `sysLib.commit`
  - d. Fetch another row by running an EGL **get next** statement

If you do not specify **hold**, the first run of step 3d fails because the cursor is no longer open.

Cursors for which you specify **hold** are not closed on a commit, but a rollback or database connect closes all cursors.

If you have no need to retain cursor position across a commit, do not specify **hold**.

### **forUpdate**

Option that lets you use a later EGL statement to replace or delete the data that was retrieved from the database.

You cannot specify **forUpdate** if you are calling a stored procedure to retrieve a result set.

### **usingKeys ... item**

Identifies a list of key items that are used to build the key-value component of the WHERE clause in an implicit SQL statement. The implicit SQL statement is used at run time if you do not specify an explicit SQL statement.

If you do not specify a **usingKeys** clause, the key-value component of the implicit statement is based on the SQL record part that is referenced in the **open** statement.

The **usingKeys** information is ignored if you specify an explicit SQL statement.

### **with #sql{ sqlStatement }**

An explicit SQL SELECT statement, which is optional if you also specify an SQL record. Leave no space between the **#sql** and the left brace.

**into ... item**

An INTO clause, which identifies the EGL host variables that receive values from the cursor or stored procedure. In a clause like this one (which is outside of a `#sql{ }` block), do not include a semicolon before the name of a host variable.

**with preparedStatementID**

The identifier of an EGL **prepare** statement that prepares an SQL SELECT or CALL statement at run time. The **open** statement runs the SQL SELECT or CALL statement dynamically. For details, see *prepare*.

**using ... item**

A USING clause, which identifies the EGL host variables that are made available to the prepared SQL SELECT or CALL statement at run time. In a clause like this one (which is outside of a `#sql{ }` block), do not include a semicolon before the name of a host variable.

*SQL record name*

Name of a record of type `SQLRecord`. Either the record name or a value for *sqlStatement* is required; if *sqlStatement* is omitted, the SQL SELECT statement is derived from the SQL record.

Examples are as follows (assuming an SQL record called *emp*):

```
open empSetId forUpdate for emp;

open x1 with
  #sql{
    select empnum, empname, empphone
    from employee
    where empnum >= :empnum
    for update of empname, empphone
  }

open x2 with
  #sql{
    select empname, empphone
    from employee
    where empnum = :empnum
  }
for emp;

open x3 with
  #sql{
    call aResultSetStoredProc(:argumentItem)
  }
```

## Default processing

The effect of an open statement is as follows by default, when you specify an SQL record:

- The open statement makes a set of rows available. Each column in the selected rows is associated with a structure item, and except for the columns that are associated with a read-only structure item, all the columns are available for subsequent update by an EGL replace statement.
- If you declare only one key item for the SQL record, the open statement selects all rows that fulfill the record-specific **default select condition**, so long as the value in the SQL table key column is greater than or equal to the value in the key item of the SQL record.
- If multiple keys are declared for the SQL record, the record-specific **default select condition** is the only search criterion, and the **open** statement retrieves all rows that meet that criterion.

- If you specify neither a record key nor a default selection condition, the **open** statement selects all rows in the table.
- The selected rows are not sorted.

The EGL **open** statement is represented in the generated code by a cursor declaration that includes an SQL SELECT or an SQL SELECT FOR UPDATE statement. The following is true by default:

- The FOR UPDATE clause (if any) does not include structure items that are read only
- The SQL SELECT statement for a particular record is similar to the following statement:

```
SELECT column01,
       column02, ...
       columnNN
INTO   :recordItem01,
       :recordItem02, ...
       :recordItemNN
FROM   tableName
WHERE  keyColumn01 = :keyItem01
FOR UPDATE OF
       column01,
       column02, ...
       columnNN
```

You may override the default by specifying an SQL statement in the EGL **open** statement.

## Error conditions

Various conditions are not valid, including these:

- You include an SQL statement that lacks a clause required for SELECT; the required clauses are SELECT, FROM, and (if you specify **forUpdate**) FOR UPDATE OF
- Your SQL record is associated with a column that either does not exist at run time or is incompatible with the related structure item
- You specify the option **forUpdate**, and your code tries to run an **open** statement against either of the following SQL records:
  - An SQL record whose only structure items are read only; or
  - An SQL record that is related to more than one SQL table.

A problem also arises in the following case:

1. You customize an EGL **open** statement for update, but fail to indicate that a particular SQL table column is available for update; and
2. The **replace** statement that is related to the **open** statement tries to revise the column.

You can solve this problem in any of these ways:

- When you customize the EGL **open** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL **replace** statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **open** and **replace** statements.

## Related concepts

“Record types and properties” on page 138



“SQL support” on page 277  
 “resultSetID” on page 867  
 “References to parts” on page 23

#### Related tasks

“Syntax diagram for EGL statements and commands” on page 884

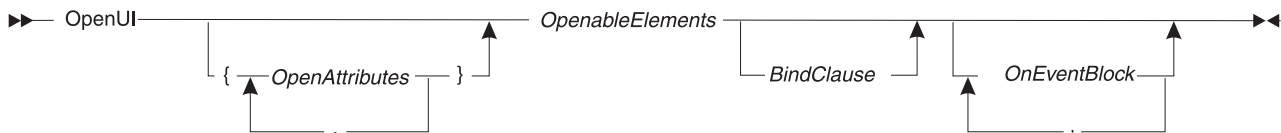
#### Related reference

“add” on page 661  
 “close” on page 669  
 “delete” on page 673  
 “EGL statements” on page 88  
 “Exception handling” on page 94  
 “execute” on page 677  
 “get” on page 687  
 “get next” on page 701  
 “get previous” on page 708  
 “I/O error values” on page 638  
 “prepare” on page 736  
 “replace” on page 738  
 “SQL item properties” on page 68  
 “terminalID” on page 1075

## openUI

The **OpenUI** statement allows the user to interact with a program whose interface is based on consoleUI. The statement defines user and program events and specifies how to respond to each.

The syntax of the OpenUI statement is as follows:



#### OpenAttributes

*OpenAttributes* defines a set of property-and-value pairs, each separated from the next by a comma, as in this example:

```
allowAppend = yes, allowDelete = no
```

Each of the properties affects the user interaction, and some overwrite a property of the consoleUI variable referenced in *OpenableElements*. The properties are as follows:

#### allowAppend

Specifies whether the user can insert data at the end of on on-screen arrayDictionary; if *yes*, the implications are as follows:

- The user inserts a row of data by moving the cursor to the arrayDictionary line which follows the last line that includes data
- The user’s action appends an element to the dynamic array that is bound to that arrayDictionary

**For variable type:** *ArrayDictionary*

**Property type:** *Boolean*

**Example:** *allowAppend = no*

**Default:** *yes; but the default is no if the openUI property **displayOnly** is set to yes*

#### **allowDelete**

Specifies whether the user can delete a row from an on-screen arrayDictionary; if *yes*, the implications are as follows:

- The user deletes a row by moving the cursor to that row and pressing the key referenced in **ConsoleLib.key\_delete**.
- The user's action deletes the related element in the dynamic array that is bound to that arrayDictionary.

**For variable type:** *ArrayDictionary*

**Property type:** *Boolean*

**Example:** *allowDelete = no*

**Default:** *yes; but the default is no if the openUI property **displayOnly** is set to yes*

#### **allowInsert**

Specifies whether the user can insert a row into an on-screen arrayDictionary; if *yes*, the implications are as follows:

- The user inserts the row by moving the cursor to an existing row and pressing the key referenced in **ConsoleLib.key\_insert**.
- The new row precedes the row that shows the cursor
- The user's action inserts an element to the dynamic array that is bound to that arrayDictionary.

**For variable type:** *ArrayDictionary*

**Property type:** *Boolean*

**Example:** *allowInsert = no*

**Default:** *yes; but the default is no if the openUI property **displayOnly** is set to yes*

#### **bindingByName**

Indicates how to bind a series of variables to a series of ConsoleFields; specifically, whether to match each variable name with a ConsoleField name. The variable name is listed in *BindClause*, and the ConsoleField name is the value in the ConsoleField name field.

**For variable type:** *ConsoleForm, ConsoleField, or Dictionary; but not arrayDictionary*

**Property type:** *Boolean*

**Example:** *bindByName = yes*

**Default:** *no*

Values are as follows:

##### **no (the default)**

Match variables and ConsoleFields by position:

- The position of each variable in the list; and
- The position of each ConsoleField in the consoleForm.

Whether consoleFields are listed explicitly in the openUI statement or are listed in a dictionary declaration, their order defines the order of consoleFields for the purpose of binding by position. (Their order also defines the tab order for user input, as noted in *ConsoleUI parts and related variables*.)

**yes**

Match variables and ConsoleFields by name.

If a consoleField is listed or is in a dictionary declaration when no matching variable is in the binding list, the user's input to the consoleField is ignored. Similarly, a binding variable that does not match any field is ignored.

At least one consoleField and variable must be bound together at run time; otherwise, an error occurs.

**color**

Specifies the color of the text in the ConsoleFields. The value overrides the color specified in the ConsoleField declaration.

**For variable type:** *ConsoleForm, ConsoleField, ArrayDictionary, or Dictionary*

**Property type:** *ColorKind*

**Example:** *color = red*

**Default:** *white*

Values are as follows:

**defaultColor or white (the default)**

White

**black**

Black

**blue**

Blue

**cyan**

Cyan

**green**

Green

**magenta**

Magenta

**red**

Red

**yellow**

Yellow

**currentArrayCount**

Specifies the number of elements that are available in the dynamic array to which the on-screen arrayDictionary is bound. If you do not specify this value, all the elements are available for use in the arrayDictionary.

**For variable type:** *ArrayDictionary*

**Property type:** *INT*

**Example:** *currentArrayCount = 4*

**Default:** *none*

**displayOnly**

Specifies whether consoleFields are displayed for viewing only. If *yes*, the user cannot modify the data, which is protected from update.

**For variable type:** *ArrayDictionary, Dictionary, ConsoleField, ConsoleForm*

**Property type:** *Boolean*

**Example:** *displayOnly = yes*

**Default:** *no*

## **help**

Specifies the text to display when the user presses the key identified in **ConsoleLib.key\_help**.

This help text is for the **openUI** command. In some cases, the text associated with the key is more context specific. For instance, each option in a menu can have its own help message.

**For variable type:** *ConsoleForm, ConsoleField, ArrayDictionary, or Dictionary*

**Primitive type:** *String*

**Example:** *help = "Update the value"*

**Default:** *Empty string*

## **helpKey**

Specifies an access key for searching the resource bundle that contains text for display when the following situation is in effect:

- The cursor is in a ConsoleUI variable (such as ConsoleForm) that is identified in *OpenableElements*; and
- The user presses the key identified in **ConsoleLib.key\_help**.

If you specify both **help** and **helpKey**, **help** is used.

**For variable type:** *ConsoleForm, ConsoleField, ArrayDictionary, or Dictionary*

**Property type:** *String*

**Example:** *helpKey = "myKey"*

**Default:** *Empty string*

The resource bundle is identified by the system variable **ConsoleLib.messageResource**, as described in *messageResource*.

## **highlight**

Specifies the special effects (if any) that are used when displaying the ConsoleField. The value overrides the equivalent value specified in the ConsoleField declaration.

**For variable type:** *ConsoleForm, ConsoleField, ArrayDictionary, or Dictionary*

**Property type:** *HighlightKind[]*

**Example:** *highlight = [reverse, underline]*

**Default:** *[noHighLight]*

Values are as follows:

### **noHighlight (the default)**

Causes no special effect. Use of this value overrides any other.

### **blink**

Has no effect.

### **reverse**

Reverses the text and background colors so that (for example) if the display has a black background with white letters, the background becomes white and the text becomes black.

**underline**

Places an underline under the affected areas. The color of the underline is the color of the text, even if the value **reverse** is also specified.

**intensity**

Specifies the strength of the displayed font.

**For variable type:** *ConsoleField, ConsoleForm, ArrayDictionary, or Dictionary*

**Property type:** *IntensityKind[]*

**Example:** *intensity = [bold]*

**Default:** *[normalIntensity]*

Values are as follows:

**normalIntensity (the default)**

Causes no special effect. Use of this value overrides any other.

**bold**

Causes the text to appear in boldface.

**dim**

Causes the text to appear with a lessened intensity, as appropriate when an input field is disabled.

**invisible**

Removes any indication that the ConsoleField is on the form.

**isConstruct**

Specifies whether the purpose of the **openUI** statement is to create selection criteria for use in an SQL statement such as SELECT.

**For variable type:** *ConsoleField, ConsoleForm, Dictionary*

**Property type:** *Boolean*

**Example:** *isConstruct = no*

**Default:** *yes*

Values are as follows:

**no (the default)**

Each ConsoleField is bound to a variable, as usual.

**yes**

The **openUI** statement must be bound to a single variable of a character type. That variable does not provide initial values for the ConsoleFields, but does receive the user's input, which is formatted for use in an SQL WHERE clause.

**maxArrayCount**

Specifies the maximum number of rows that can be in the dynamic array bound to the on-screen arrayDictionary. After the maximum is reached, the user is unable to insert more rows.

**For variable type:** *ArrayDictionary*

**Property type:** *INT*

**Example:** *maxArrayCount = 20*

**Default:** *none*

**setInitial**

Specifies whether the initial value of a `ConsoleField` (as defined in the `consoleForm` declaration) is displayed until the user modifies that value. (You specify the initial value by setting the **initialValue** field of `ConsoleField`.)

**For variable type:** *ConsoleField, ConsoleForm, Dictionary, ArrayDictionary*

**Property type:** *Boolean*

**Example:** *setInitial = yes*

**Default:** *no*

If the value of **setInitial** is **no**, the values of the bound variables are fetched and displayed initially.

#### *OpenableElements*

The `ConsoleUI` variables on which the **openUI** statement can act:

- `ConsoleForm`
- A `consoleField` or one of these:
  - A list of `consoleFields`, each separated from the next by a comma
  - A dictionary that is declared in a `consoleForm` and refers to a set of `consoleFields` in that `consoleForm`
  - An `arrayDictionary` that is declared in a `consoleForm` and refers to a set of `consoleField` arrays in that `consoleForm`.
- `Menu`
- `Prompt`
- `Window`

#### *BindClause*

The list of primitive variables, records, or arrays that are bound to the `ConsoleUI` variables. Characteristics of the binding variables depend on the characteristics of the `consoleUI` variable on which the **openUI** statement acts:

- For a `consoleField`, you can specify a primitive variable.
- For an on-screen `arrayDictionary`, you can specify an array of records, one element per row in the `arrayDictionary`; and if each row in the `arrayDictionary` represents a single value, you can specify an array of primitive variables.
- For a dictionary or a list of `consoleFields`, you can specify a list of primitive variables. Alternatively, you can specify an array of records, with each element containing a series of fields that are bound to the `consoleFields`. This alternative is equivalent to binding a dynamic array with an on-screen `arrayDictionary` that has only one row; you can append, insert, or delete a record to change the dynamic array, and in any case only one record is displayed at a time.
- For a `prompt`, you can specify a primitive field that receives the user's response.

For details on binding, see the section on *OnEventBlock* (later), as well as *ConsoleUI parts and related variables*.

#### *OnEventBlock*

An *event block* is a programming structure that includes 0 to many *event handlers*, each of which contains the code that you've written for responding to a particular event. An event handler begins with an `OnEvent` header:

```
OnEvent(eventKind: eventQualifiers)
```

*eventKind*

One of several events. Valid values are described in “Event types.”

*eventQualifier*

Data that further defines the event. Such data might be the ConsoleField entered or the keystroke pressed.

The EGL statements that respond to a given event are between the OnEvent header and the next OnEvent header (if any), as shown in a later example. However, you cannot include a reference variable in the OnEvent block unless that variable was declared as a program global.

The user continually interacts with the program, and the program runs an event handler when the event occurs that is associated with that event handler. If the purpose of the **openUI** statement is to display a prompt, however, the user-program interaction is less like a loop:

1. An event handler (potentially one of several) traps a user keystroke and responds
2. The **openUI** statement ends

No event block is available for a window.

Consider the following example for guiding the interaction between the user and a ConsoleForm:

```
openUI {bindingByName=yes}
  activeForm
  bind firstName, lastName, ID
  OnEvent(AFTER_FIELD:"ID")
    if (employeeID == 700)
      firstName = "Angela";
      lastName = "Smith";
    end
end
```

That code acts as follows:

- Opens the *active ConsoleForm* (which is the consoleForm that was most recently displayed in the active window);
- Binds a set of primitive variables to each of the ConsoleFields; and
- Specifies that after the user types a value in *employeeID* and leaves that ConsoleField, EGL places strings in two other variables.

Consider these details about the preceding example:

- The cursor starts in the first of the listed consoleFields; but should start in the ID consoleField so that the user’s input in the other consoleFields is not wiped out by the event handler.
- The event handler updates the variables that are bound to the firstName and lastName consoleFields but does not cause those values to be displayed until the cursor enters those fields. You might want to display the values earlier.

You can end an **openUI** statement by issuing an **exit** statement of the form **exit openUI**.

## Event types

ConsoleUI supports the following events:

#### **BEFORE\_OPENUI**

The EGL runtime begins to run the **OpenUI** statement. This event is available for all ConsoleUI variables other than those based on Window.

#### **AFTER\_OPENUI**

The EGL runtime is about to stop running the **OpenUI** statement. This event is available for all ConsoleUI variables other than those based on Window.

#### **ON\_KEY:***(ListOfStrings)*

The user has pressed a specific key, as indicated by a string such as "ESC", "F2", or "CONTROL\_W". You can identify multiple keys by separating one string from the next, as in this example:

```
ON_KEY:("a", "ESC")
```

This event is available in an ArrayDictionary, a ConsoleField, a Menu, or a Prompt.

#### **BEFORE\_FIELD:***(ListOfStrings)*

The user has moved the cursor into the specified ConsoleField, as indicated by a string that matches the value of the ConsoleField name field. You can identify multiple consoleFields in the same consoleForm by separating one string from the next, as in this example:

```
BEFORE_FIELD("field01", "field02")
```

#### **AFTER\_FIELD:***(ListOfStrings)*

The user has moved the cursor out of the specified ConsoleField, as indicated by a string that matches the value of the ConsoleField name field. You can identify multiple consoleFields in the same consoleForm by separating one string from the next, as in this example:

```
AFTER_FIELD("field01", "field02")
```

#### **BEFORE\_DELETE**

In relation to an on-screen arrayDictionary, the user has pressed the key specified in **ConsoleLib.key\_deleteLine**, but the EGL runtime has not yet deleted the row. The program can invoke **consoleLib.cancelDelete** to avoid deleting the row.

#### **BEFORE\_INSERT**

In relation to an on-screen arrayDictionary, the user has pressed the key specified in **ConsoleLib.key\_insertLine**, but the EGL runtime has not yet inserted a row. The program can invoke **consoleLib.cancelInsert** to avoid inserting the row.

#### **BEFORE\_ROW**

In relation to an on-screen arrayDictionary, the user has moved the cursor into a row.

#### **AFTER\_DELETE**

In relation to an on-screen arrayDictionary, the user has pressed the key specified in **ConsoleLib.key\_deleteLine**, and the EGL runtime has deleted a row.

#### **AFTER\_INSERT**

In relation to an on-screen arrayDictionary, the user has pressed the key specified in **ConsoleLib.key\_insertLine**; the EGL runtime has inserted a row; and the cursor leaves the row that was inserted.

The user can edit the row before committing changes to a database, as typically happens in the AFTER\_INSERT handler.



## AFTER\_ROW

The user has moved the cursor from a row in an on-screen arrayDictionary.

## MENU\_ACTION:(ListOfStrings)

The user has selected a menuItem, as indicated by a string that matches the value of the menuItem name field. You can identify multiple menuItems by separating one string from the next, as in this example:

```
MENU_ACTION:("item01", "item02")
```

## isConstruct

When the property **isConstruct** = *yes*, the text placed in the variable bound to the **openUI** command is specially formatted, as shown in this example:

1. An openUI statement acts on a ConsoleForm of three ConsoleFields (*employee*, *age*, and *city*), and each field is associated with an SQL table column of the same name.

You associate a consoleField with an SQL table column by setting the consoleField property **SQLColumnName**; and you must set the consoleField property **dataType**, as noted in *ConsoleField Properties and Fields*.

2. The user acts as follows:

- Leaves the *employee* field blank
- Types this in the *age* field:  
    > 25
- Types this in the *city* field:  
    = 'Sarasota'

3. When the user leaves the on-screen variable on which the openUI statement acts, the bound variable receives the following content:

```
AGE > 28 AND CITY = 'Sarasota'
```

As shown, EGL places the operator AND between each clause that the user provides.

The next table shows valid user input and the resulting clause. The phrase *simple SQL types* refers to SQL types that are neither structured nor LOB-like types.

Symbol	Definition	Supported data types	Example	Resulting clause (for a character column named C)	Resulting clause (for a numeric column named C)
=	Equal to	Simple SQL Types	=x, ==x	C = 'x'	C = x
>	Greater than	Simple SQL Types	>x	C > 'x'	C > x
<	Less than	Simple SQL Types	<x	C < 'x'	C < x
>=	Not less than	Simple SQL Types	>=x	C >= 'x'	C >= x
<=	Not greater than	Simple SQL Types	<=x	C <= 'x'	C <= x
<> or !=	Not equal to	Simple SQL Types	<>x or !=x	C != 'x'	C != x
..	Range	Simple SQL Types	x.y or x..y	C BETWEEN 'x' AND 'y'	C BETWEEN x AND y

Symbol	Definition	Supported data types	Example	Resulting clause (for a character column named C)	Resulting clause (for a numeric column named C)
*	Wildcard for String (as described in next table)	CHAR	*x or x* or *x*	C MATCHES '*x'	not applicable
?	Single character wildcard (as described in next table)	CHAR	?x, x?, ?x?, x??	C MATCHES '?x'	not applicable
	Logical Or	Simple SQL Types	x y	C IN ('x', 'y')	C IN (x, y)

The equal sign (=) can mean IS NULL; and the not-equal sign (!= or <>) can mean IS NOT NULL.

A MATCHES clause results from the user's specifying one of the wildcard characters described in the next table.

Symbol	Effect
*	Matches zero or more characters.
?	Matches any single character.
[ ]	Matches any enclosed character.
- (hyphen)	When used between characters inside brackets, a hyphen matches any character in the range between and including the two characters. For example, [a-z] matches any lowercase letter or special character in the lower case range.
^	When used in brackets, an initial caret matches any character not included within the brackets. For example, [^abc] matches any character except a, b, or c.
\	Is the default escape character; the next character is a literal. Allows any of the wildcard characters to be included in the string without having the wildcard effect.
Any other character outside of brackets	Must match exactly.

#### Related concepts

"Console user interface" on page 207

#### Related reference

"EGL library ConsoleLib" on page 886

"ConsoleUI parts and related variables" on page 209

"ConsoleUI screen options for UNIX" on page 213

"Text UI program in EGL source format" on page 844

#### Related tasks

"Creating an interface with ConsoleUI" on page 208

**prepare**

The EGL **prepare** statement specifies an SQL PREPARE statement, which optionally includes details that are known only at run time. You run the prepared SQL statement by running an EGL **execute** statement or (if the SQL statement returns a result set) by running an EGL **open** or **get** statement.

►► prepare — *preparedStatementID* — from *stringExpression* — ; ►►  
 — for SQL record name —

```
preparedStatementID
```

An identifier that relates the **prepare** statement to the **execute**, **open**, or **get** statement.

*stringExpression*

A *string expression* that contains a valid SQL statement.

SQL record name

Name of an SQL record. Specifying this name has two benefits:

- The EGL editor provides a dialog to derive an SQL statement based on your specifications
- After the **prepare** statement runs, you can test the record name against an I/O error value to determine whether the statement succeeded, as in the following example:

```
try
  prepare prep01 from
    "insert into " + aTableName +
    "(empnum, empname) " +
    "value ?, ?"
  for empRecord;
```

```
onException
  if empRecord is unique
    myErrorHandler(8);
  else
    myErrorHandler(12);
  end
end
```

Another example is as follows:

```
myString =
    "insert into myTable " + "(empnum, empname) " +
    "value ?, ?";
```

```
try
    prepare myStatement
    from myString;
onException
    myErrorHandler(12);    // exits the program
end
```

```
try
    execute myStatement
    using :myRecord.empnum,
        :myRecord.empname;
onException
    myErrorHandler(15);
end
```

As shown in the previous examples, the developer can use a question mark (?) where a host variable should appear. Then, the name of the host variable used at run time is placed in the using clause of the **execute**, **open**, or **get** statement that runs the prepared statement.

A **prepare** statement that acts on a row of a result set may include a phrase of the format *where current of resultSetIdentifier*. This technique is valid only in the following situation:

- The phrase is coded in a literal; and
- The result set is open when the **prepare** statement runs.

An example is as follows:

```
prepare prep02 from
  "update myTable " +
  "set empname = ?, empphone = ? where current of x1" ;

execute prep02 using empname, empphone ;
freeSQL prep02;
```

For a example of how parentheses affect the use of a plus (+) sign, see *Expressions*.

#### Related concepts

"References to parts" on page 23

"Record types and properties" on page 138

"SQL support" on page 277

#### Related reference

"add" on page 661

"close" on page 669

"delete" on page 673

"Exception handling" on page 94

"execute" on page 677

"Expressions" on page 591

"freeSQL" on page 687

"get" on page 687

"get next" on page 701

"get previous" on page 708

"I/O error values" on page 638

"EGL statements" on page 88

"open" on page 722

"replace" on page 738

"SQL item properties" on page 68

"Text expressions" on page 601

"Syntax diagram for EGL statements and commands" on page 884

## print

The EGL **print** statement adds a print form to a runtime buffer, as described in *Print forms*.

► print *printFormName* ;

### printFormName

Name of a print form that is visible to the program. For details on visibility, see *References to parts*.

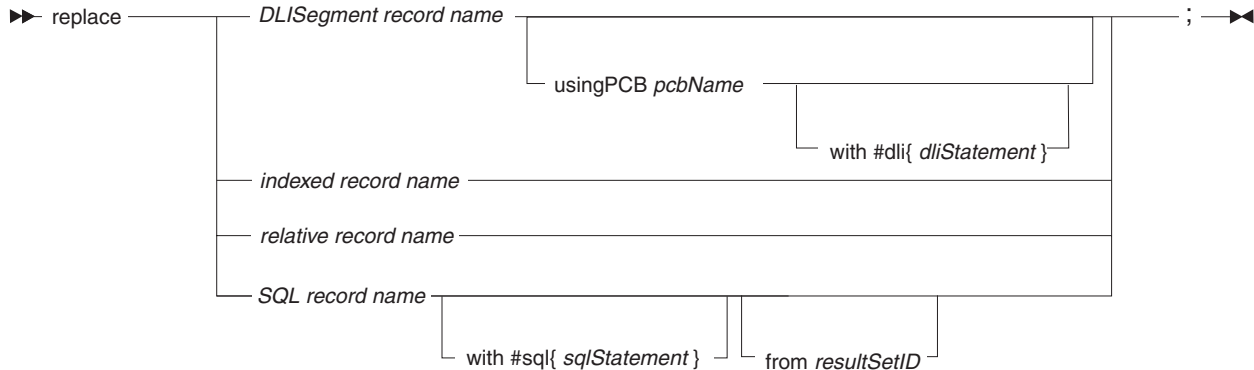
### Related concepts

"Print forms" on page 186

"References to parts" on page 23

## replace

The EGL **replace** statement puts a changed record into a file or database.



### *record name*

Name of the I/O object: a DLISegment, indexed, relative, or SQL record.

### **usingPCB** *pcbName*

Option allowing you to specify the name of a PCB, as defined in your PSB record, to use instead of the default PCB.

### **with #dli{** *dliStatement* **}**

Option that allows an explicit DL/I REPL statement, as described in *#dli directive*. Leave no space between **#dli** and the left brace.

### **with #sql{** *sqlStatement* **}**

An explicit SQL UPDATE statement. Leave no space between **#sql** and the left brace.

### **from** *resultSetID*

ID that ties the **replace** statement to a **get** or **open** statement run earlier in the same program. For details, see *resultSetID*.

The following example shows how to read and replace a file record:

```
emp.empnum = 1;           // sets the key in record emp

try
  get emp forUpdate;
onException
  myErrorHandler(8); // exits the program
end

emp.empname = emp.empname + " Smith";

try
  replace emp;
onException
  myErrorHandler(12);
end
```

Details on the **replace** statement depend on the record type. For details on SQL processing, see *SQL record*.

### DLISegment record

The **replace** statement generates a DL/I REPL statement. In DL/I, you must get and hold a segment before overwriting it. The EGL keywords **get...forUpdate**, **get next...forUpdate**, and **get next inParent...forUpdate** will all hold the requested segment for replacement.

The following example overwrites an order from the customer database:

```
//create instances of the records
myCustomer CustomerRecord;
myLocation LocationRecord;
myOrder OrderRecord;

//build a segment search argument
myCustomer.customerNo = "5001";
myLocation.locationNo = "22";
myOrder.orderDateNo = "20050730A003";

//hold the requested order segment
try
  get myOrder forUpdate;
onException
  myErrorHandler(2);
end

//update the information
changeOrder(myOrder);

//rewrite the order
try
  replace myOrder;
onException
  myErrorHandler(8);
end
```

### Indexed or relative record

If you want to replace an indexed or relative record, you must issue a **get** statement for the record with the **forUpdate** option, then issue the **replace** statement with no intervening I/O operation against the same file. After you invoke the **replace** statement, the effect of the next I/O operation on the same file is as follows:

- If the next I/O operation is a **replace** statement on the same EGL record, the record is changed in the file
- If the next I/O operation is a **delete** statement on the same EGL record, the record in the file is marked for deletion
- If the next I/O operation is a **get** statement on a record in the same file and includes the **forUpdate** option, a subsequent **replace** or **delete** statement is valid on the newly read file record
- If the next I/O operation is a **get** statement on the same EGL record (with no **forUpdate** option) or is a **close** statement on the same file, the file record is released without change

For details on the **forUpdate** option, see *get*.

### SQL record

In the case of SQL processing, the EGL **replace** statement results in an SQL UPDATE statement in the generated code.

You must retrieve a row for subsequent replacement, in either of two ways:

- Issue a **get** statement (with the `forUpdate` option) to retrieve the row; or
- Issue an **open** statement to select a set of rows, then invoke a **get next** statement to retrieve the row of interest.

**Error conditions:** The following conditions are among those that are not valid when you use a **replace** statement:

- You specify an SQL statement of a type other than UPDATE
- You specify some but not all clauses of an SQL UPDATE statement
- You do not specify a *resultSetID* value when one is necessary; for details, see *resultSetID*
- You specify (or accept) an UPDATE statement that has one of these characteristics--
  - Updates multiple tables
  - Is associated with a column that either does not exist or is incompatible with the related host variable
- You use an SQL record as an I/O object, and all the record items are read only

The following situation also causes an error:

- You customize an EGL **get** statement with the `forUpdate` option, but fail to indicate that a particular SQL table column is available for update; and
- The **replace** statement that is related to the **get** statement tries to revise the column.

You can solve the previous mismatch in any of these ways:

- When you customize the EGL **get** statement, include the column name in the SQL SELECT statement, FOR UPDATE OF clause; or
- When you customize the EGL **replace** statement, eliminate reference to the column in the SQL UPDATE statement, SET clause; or
- Accept the defaults for both the **get** and **replace** statements.

**Implicit SQL statement:** By default, the effect of a **replace** statement that writes an SQL record is as follows:

- As a result of the association of record items and SQL table columns in the record declaration, the generated code copies the data from each record item into the related SQL table column
- If you defined a record item to be read only, the value in the column that corresponds to that record item is unaffected

The SQL statement has these characteristics by default:

- The SQL UPDATE statement does not include record items that are read only
- The SQL UPDATE statement for a particular record is similar to the following statement:

```
UPDATE tableName
SET column01 = :recordItem01,
    column02 = :recordItem02,
    .
    .
    .
    columnNN = :recordItemNN   WHERE CURRENT OF cursor
```

### Related concepts

"Record types and properties" on page 138  
"References to parts" on page 23  
"resultSetID" on page 867  
"Run unit" on page 866  
"SQL support" on page 277

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

### Related reference

"#dli directive" on page 325  
"add" on page 661  
"CICS-related considerations" on page 530  
"close" on page 669  
"delete" on page 673  
"EGL statements" on page 88  
"Exception handling" on page 94  
"execute" on page 677  
"get" on page 687  
"get next" on page 701  
"get" on page 687  
"get previous" on page 708  
"I/O error values" on page 638  
"open" on page 722  
"prepare" on page 736  
"set" on page 742  
"SQL item properties" on page 68  
"terminalID" on page 1075

## return

The EGL **return** statement exits from a function and optionally returns a value to the invoking function.



### *returnValue*

An item, literal, or constant that is compatible with the **returns** specification in the EGL function declaration.

Although an item must correspond in all ways to the **returns** specification, the rules for literals and constants are as follows:

- A numeric literal or constant can be returned only if the primitive type in the **returns** specification is a numeric type
- A literal or constant that includes only single-byte characters can be returned only if the primitive type in the **returns** specification is CHAR or MBCHAR
- A literal or constant that includes only double-byte characters can be returned only if the primitive type in the **returns** specification is DBCHAR



- A literal or constant that includes a combination of single- and double-byte characters can be returned only if the primitive type in the **returns** specification is MBCHAR
- A literal or constant cannot be returned if the primitive type in the **returns** specification is HEX

A function that includes a **returns** specification must terminate with a **return** statement that includes a value. A function that lacks a **returns** specification may terminate with a **return** statement, which must not include a value.

The **return** statement gives control to the first statement that follows invocation of the function, even if the statement is in an **OnException** clause in a try block.

## set

The following sections describe the effect of an EGL **set** statement:

- “Effect on a record (or fixed record) as a whole”
- “Effect on a form as a whole” on page 744
- “Effect on a field in any context” on page 745
- “Effect on a field in a text form” on page 746

### Effect on a record (or fixed record) as a whole

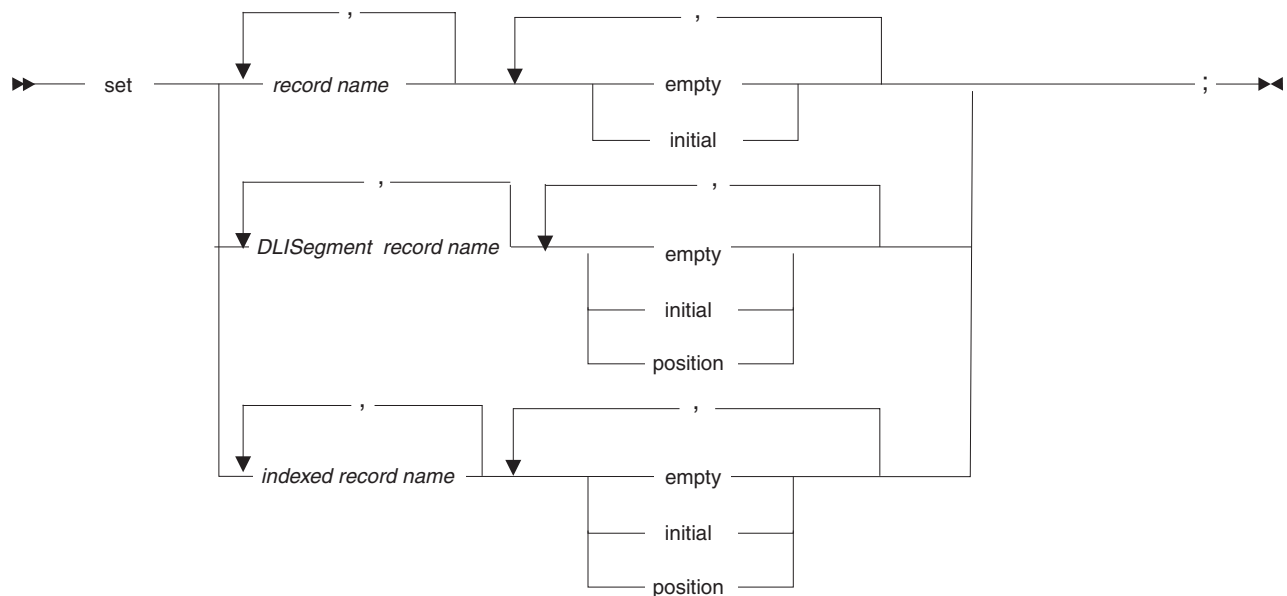
The next table describes the **set** statements that affect a record as a whole, a fixed record as a whole, or an array of either.

Format of set statement	Effect
set record empty	<p>Empties each of the elementary fields. For a record, each subordinate record is emptied, as is each subordinate of those subordinates, and so on. For a fixed record (which may itself be in a record), the elementary fields are at the lowest level of the fixed structure.</p> <p>The effect on each elementary field depends on the primitive type of that field:</p> <ul style="list-style-type: none"> <li>• For a field of type ANY, the <b>set</b> statement initializes the field according to the field’s current type; and if the field is of type ANY and has no other type, the <b>set</b> statement has no effect</li> <li>• For details on fields of other types, see <i>Data initialization</i></li> </ul>
set record initial	<p>Resets the field values to those specified by the <b>value</b> property at development time, as is possible for a record or fixed record that is declared in a pageHandler or form. A value set by assignment is never reinstated.</p> <p>If the <b>value</b> property has no value or if the record is not in a pageHandler or form, the effect of <i>set record initial</i> is the same as the effect of <i>set record empty</i>, with one exception: for a field that is of type ANY, the <b>set</b> statement removes any type specification other than ANY.</p>

Format of set statement	Effect
set record position	Establishes position in the VSAM file associated with a fixed record of type indexedRecord (described later), or establishes position in DL/I database associated with a record of type DLISegment (also described later).  This set-statement format is not available for an array.

You can combine statement formats by inserting a comma to separate the options. For a given record, the options take effect in the order in which they appear in the **set** statement. Also, you can specify multiple records by inserting a comma to separate one from the next.

The syntax diagram is as follows:



#### *record name*

Name of a record or fixed record of any type. You can specify an array.

#### *indexed record name*

Name of a fixed record of type indexedRecord. You can specify an array only if you do not include *set record position*.

#### **empty**

As described in the previous table.

#### **initial**

As described in the previous table.

#### **position**

Establishes a file or database position based on the *set value*, which is the key value in an indexed record or DLISegment record. The overall effect depends on the next input or output operation that your code performs against the same record:

- If the next operation is an EGL **get next** statement, that statement reads the first file record or segment that has a key value equal to or greater than the set value. If no such record or segment exists, the result of the **get next** statement is **endOfFile**.
- If the next operation after *set record position* is an EGL **get previous** statement, that statement reads the first file record that has a key value equal to or less than the set value. If no such record exists, the result of **get previous** is **endOfFile**. (The **get previous** statement does not support DL/I access.)
- Any other operation after *set record position* resets the file position, and the *set record position* has no effect.

For indexed records, if the set value is filled with hexadecimal FF values, the following is true:

- The *set record position* establishes a file position after the last record in the file
- If the next operation is a **get previous** statement, the last record in the file is retrieved

The *set record position* performs no I/O. For DL/I databases, it sets a flag telling the system that a subsequent GN (get next) call should execute the default GU (get unique) call implied by the key fields in the various DLI<sub>Segment</sub> records. Performing a *set record position* before entering a loop with a **get next** statement may be more efficient than performing an explicit **get** before entering the loop.

## Effect on a form as a whole

The next table describes the **set** statements that affect a form as a whole.

Format of set statement	Effect
set form alarm	For text forms only; sounds an alarm the next time that a <b>converse</b> statement presents the form.
set form empty	Empties the value of each field in the form, clearing any content. The effect on a given field depends on the primitive type: <ul style="list-style-type: none"> <li>• For a field of type ANY, the <b>set</b> statement initializes the field according to the field's current type; and if the field is of type ANY and has no other type, the <b>set</b> statement has no effect</li> <li>• For details on fields of other types, see <i>Data initialization</i></li> </ul>
set form initial	Resets each form field to its originally defined state, as expressed in the form declaration. Changes that were made by the program are canceled. for a field that is of type ANY, the <b>set</b> statement removes any type specification other than ANY.
set form initialAttributes	Resets each form field to its originally defined state, as expressed in the form declaration. The content of the field is not affected, neither (in the case of a field of type ANY) is the type affected.

You can combine statement formats by inserting a comma to separate options such as **empty** and **alarm**. Also, you can specify multiple forms by inserting a comma to separate one form from the next.

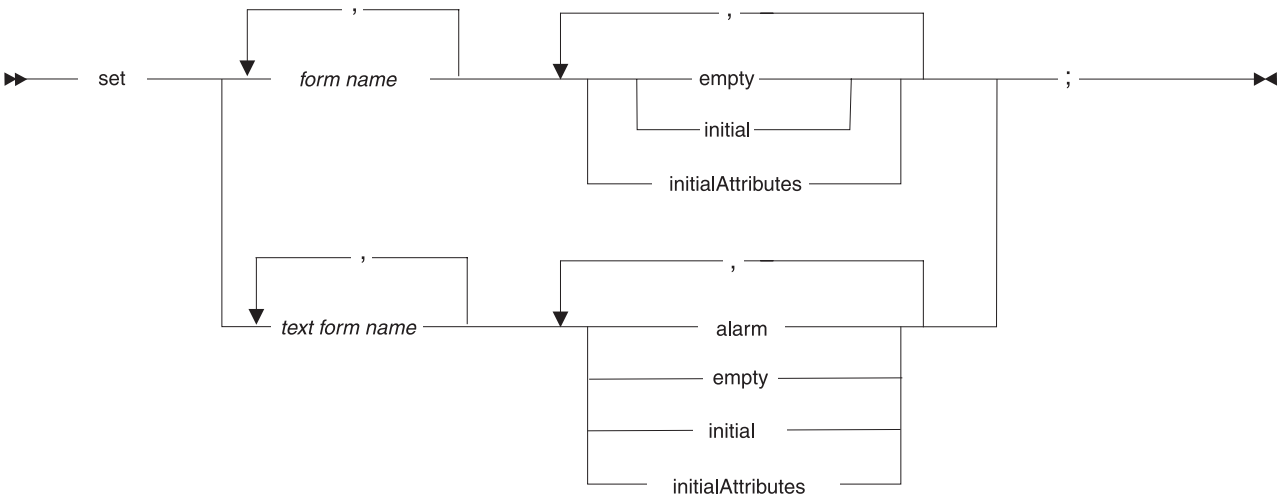
Of the following formats, you can choose one or none:

- *set form empty*
- *set form initial*

Of the following formats, you can choose one, both, or none:

- *set form alarm* (available only for text forms)
- *set form initialAttributes*

The syntax diagram is as follows:



*form name*

Name of a form of type *text* or *print*, as described in *Form part*.

*text form name*

Name of a form of type *text*, as described in *Form part*.

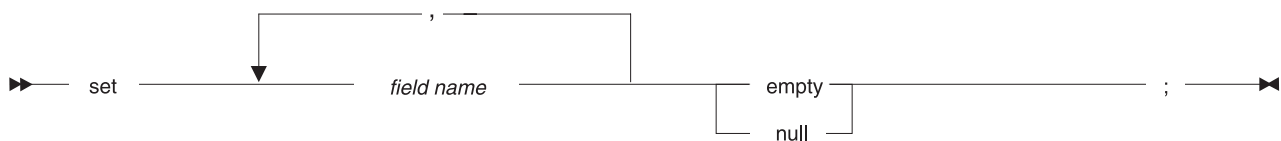
The options are as described in the previous table.

### Effect on a field in any context

The next table describes the format of the **set** statement that affects a field in any context.

Format of set statement	Effect
set field empty	<p>Empties the field or (for a fixed field that has a substructure) empties every subordinate, elementary field.</p> <p>The effect depends on the primitive type of a field:</p> <ul style="list-style-type: none"> <li>For a field of type ANY, the <b>set</b> statement initializes the field according to the field's current type; and if the field is of type ANY and has no other type, the <b>set</b> statement has no effect</li> <li>For details on fields of other types, see <i>Data initialization</i></li> </ul>
set field null	<p>Nulls the field, if doing so is valid. For details on when the operation is valid, see <i>itemsNullable</i>.</p>

The syntax diagram is as follows:



*field name*

Name of the field.

You may select one or the other option, and each is described in the previous table.

### Effect on a field in a text form

The next table describes the **set** statements that affect a field or an array of fields in a text form. A given **set** statement can combine options only in a particular set of ways, as described later. Many of the actions described are dependent on the device where the text form is displayed. It is recommended that you test your output on each of the devices that you are supporting.

Format of set statement	Effect
set field blink	Causes the text to blink repeatedly. This option is available only in COBOL programs.
set field bold	Cause the text to appear in boldface.
set field cursor	<p>Positions the cursor in the specified field.</p> <p>If the field identifies an array and has no occurs value, the cursor is positioned at the first array element by default.</p> <p>If your program runs multiple statements of the format <i>set field cursor</i>, the last is in effect when the <b>converse</b> statement runs.</p>

Format of set statement	Effect
set field defaultColor	Sets the field-specific <b>color</b> property to <i>defaultColor</i> , which means that other conditions determine the displayed color. For details, see <i>Field-presentation properties</i> .
set field dim	Causes the field to be appear in lower intensity than normal. Use this effect to deemphasize field contents. In COBOL environments, the clause has the same effect as <i>set field normalIntensity</i> , causing the field to be visible, without boldface.
set field empty	Initializes the value of the field, clearing any content. The effect on a given field depends on the primitive type, as described in <i>Data initialization</i> .
set field full	<p>Sets an empty, blank, or null field to a series of identical characters before the form is presented:</p> <ul style="list-style-type: none"> <li>The character is an asterisk (*) if the field property <b>fillCharacter</b> is the following value (which is also the default value for <b>fillCharacter</b>): <ul style="list-style-type: none"> <li>0 for fields of type HEX</li> <li>space for fields of a numeric type</li> <li>empty string for other fields</li> </ul> </li> <li>If <b>fillCharacter</b> is not set as described, the character is identical to the value of <b>fillCharacter</b>.</li> </ul> <p>The on-form characters are returned to the program only if the modified data tag for the field is set, as described in <i>Modified data tag and property</i>. A user who changes the field must remove all the on-field characters to prevent their return to the program.</p> <p>Use of <i>set field full</i> has an effect only if the form group is generated with the build descriptor option <i>setFormItemFull</i>.</p> <p>A field of type MBCHAR is considered to be empty if it contains all single-byte spaces. In relation to such fields, <i>set field full</i> assigns a series of single-byte characters.</p>
set field initial	Resets the field to its originally defined state, independent of any changes made by the program
set field initialAttributes	Resets the field to its originally defined state, without using the <b>value</b> property (which specifies the current content of the field)
set field invisible	Makes the field text invisible

Format of set statement	Effect
set field masked	Appropriate for password fields. If the text form is presented by a Java program, an asterisk is displayed instead of any non-blank character that the user types into an input field. If the text form is presented by a COBOL program, this option makes the field text invisible.
set field modified	Sets the modified data tag, as described in <i>Modified data tag and property</i> .
set field noHighlight	Eliminates the special effects of blink, reverse, and underline.
set field normal	Resets the fields as described in relation to the following formats: <ul style="list-style-type: none"> <li>• Set field normalIntensity</li> <li>• Set field unmodified</li> <li>• Set field unprotected</li> </ul> For details, see the next table.
set field normalIntensity	Sets the field to be visible, without boldface.
set field protect	Sets the field so that the user cannot overwrite the value in it. See also <i>set field skip</i> .
set field reverse	Reverses the text and background colors, so that (for example) if the display has a dark background with light letters, the background becomes light and the text becomes dark.
set field <i>selectedColor</i>	Sets the field-specific <b>color</b> property to the value you specify. The valid values for <i>selectedColor</i> are as follows: <ul style="list-style-type: none"> <li>• black</li> <li>• blue</li> <li>• green</li> <li>• pink</li> <li>• red</li> <li>• turquoise</li> <li>• white</li> <li>• yellow</li> </ul> In COBOL environments, the value of <i>black</i> is implemented as <i>defaultColor</i> , which means that other conditions determine the displayed color. For details, see <i>Field-presentation properties</i> .
set field skip	Sets the field so that the user cannot overwrite the value in it. In addition, the cursor skips the field in either of these cases: <ul style="list-style-type: none"> <li>• The user is working on the previous field in the tab order and either presses <b>Tab</b> or fills that previous field with content; or</li> <li>• The user is working on the next field in the tab order and presses <b>Shift Tab</b>.</li> </ul>

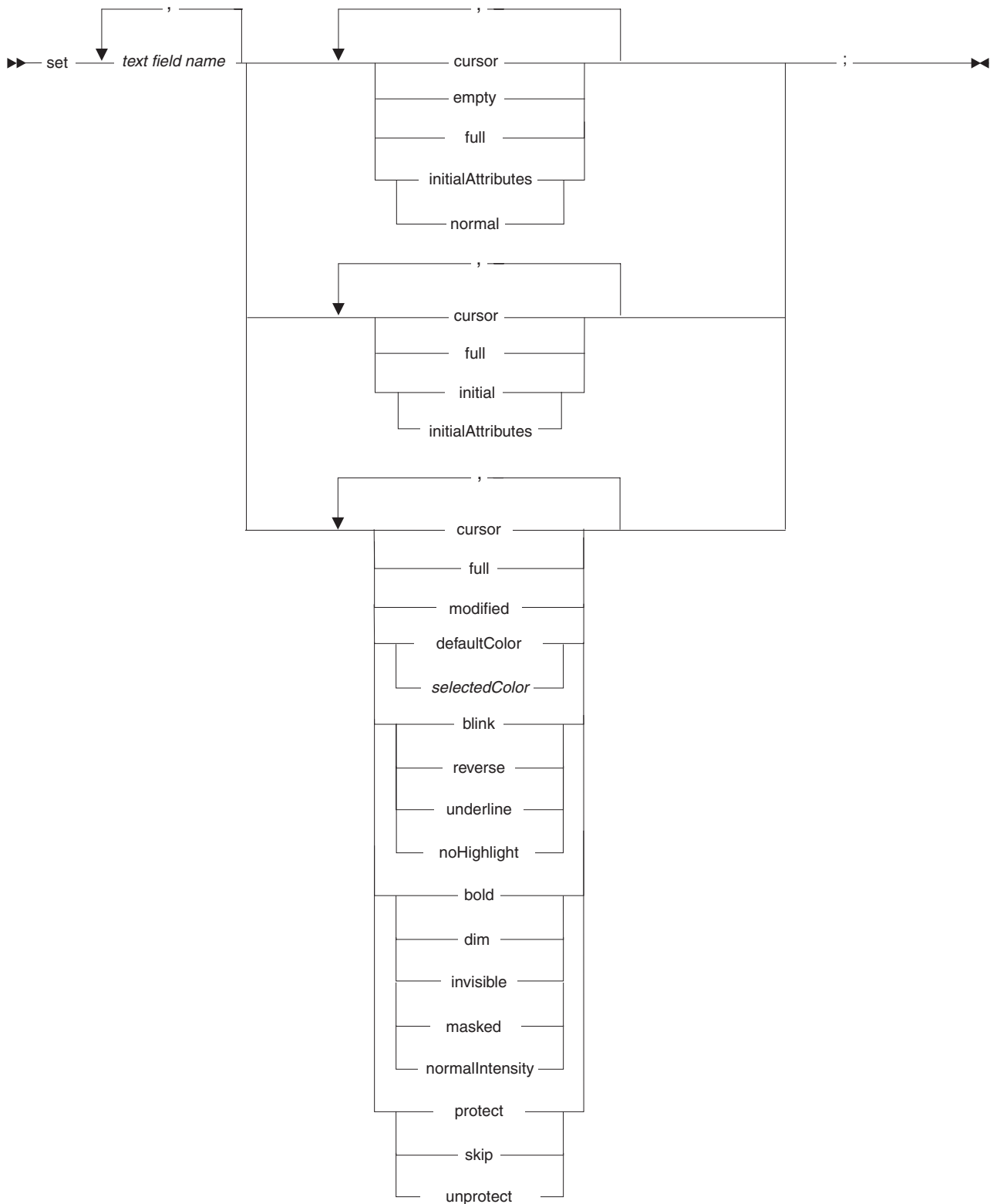
Format of set statement	Effect
set field underline	Places an underline at the bottom of the field.
set field unprotect	Sets the field so that the user can overwrite the value in it.

You can combine statement formats, inserting a comma to separate options such as **cursor** and **full**, in any of three ways:

1. You can construct a **set** statement as follows--
  - Choose one or none of these field-attribute formats:
    - *set field initialAttributes*
    - *set field normal*
  - Choose any number of the next formats:
    - *set field cursor*
    - *set field empty*
    - *set field full*
2. Second, you can construct a **set** statement from any number of the next formats:
  - *set field cursor*
  - *set field full*
  - *set field initial* or *set field initialAttributes*
3. Last, you can construct a **set** statement as follows--
  - Choose any number of the next formats:
    - *set field cursor*
    - *set field full*
    - *set field modified*
  - Choose one or none of the color formats:
    - *set field defaultColor*
    - *set field selectedColor*
  - Choose one or none of the highlight formats:
    - *set field blink*
    - *set field reverse*
    - *set field underline*
    - *set field noHighlight*
  - Choose one or none of the intensity formats:
    - *set field bold*
    - *set field dim*
    - *set field invisible*
    - *set field masked*
    - *set field normalIntensity*
  - Choose one or none of the protection formats:
    - *set field protect*
    - *set field skip*
    - *set field unprotect*



The syntax diagram is as follows:



*field name*

Name of the field in a text form. The name may refer to an array of fields.

The options are as described in the previous table.

#### Related concepts

“Form part” on page 184

“Modified data tag and modified property” on page 191

“Syntax diagram for EGL statements and commands” on page 884

#### Related reference

“Data initialization” on page 564

“EGL statements” on page 88

“Field-presentation properties” on page 67

“get next” on page 701

“get previous” on page 708

“itemsNullable” on page 482

## show

The **show** statement presents a text form (including forms buffered with the **display** statement) or processes a VGUI record from a main program:

1. Commits recoverable resources, closes files, and releases locks
2. Optionally, passes a basic record for use by the program that is specified in the **show** statement’s returning clause (if any)
3. Ends the first program
4. Presents the text form or processes the VGUI record

The **show** statement is not available in a called program.

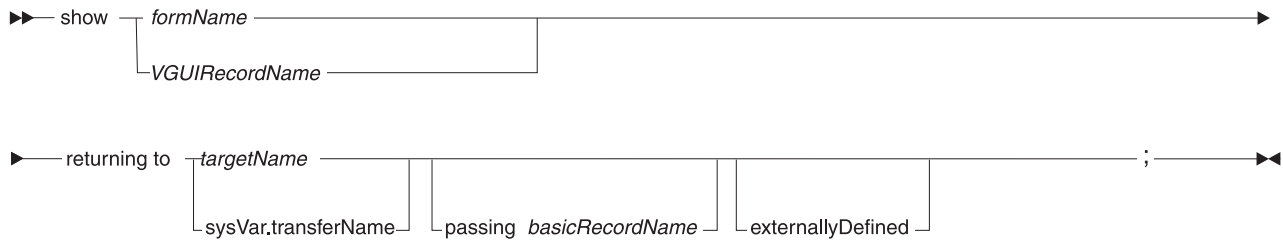
If you are processing a text form, the following statements apply:

- If you do not include a **returning** clause, the operation ends when the text form is displayed.
- If you include a **returning** clause, a specified program receives control after the user presses an event key. The form is assigned to the receiving program’s **inputForm** property, and the passed record (unchanged by user input) is assigned to the receiving program’s *input record*.

If you are processing a VGUI record, the following statements apply:

- If you do not include a **returning** clause, the operation ends when the Web page is displayed.
- If you include a **returning** clause, a specified program of type `VGWebTransaction` receives control after the user submits a form. The form data is assigned to the receiving program’s *input Ulrecord*; and the passed basic record (unchanged by user input) is assigned to the receiving program’s *input record*.

Passing the record is optional.



#### *formPartName*

Name of a text form that is visible to the program. For details on visibility, see *References to parts*. If you include a returning clause in the statement, the text form must be equivalent to the text form specified in the **inputForm** property of the program being invoked.

#### *VGUIRecordName*

Name of the VGUIRecord to present to the user.

#### *targetName*

Identifier of the program that is invoked after the user submits the text form or Web page. If the target program is on CICS, use the transaction ID; otherwise, use the program name.

#### **sysVar.transferName**

A system variable that contains the identifier of the program or transaction to be invoked. Use this variable to set the identifier at run time.

#### *basicRecordName*

Name of a record of type `basicRecord`. The content is assigned to the receiving program's *input record*.

#### **externallyDefined**

An indicator that you are returning control to a CICS-based program defined outside of EGL or VisualAge Generator. This indicator is available only if you set the project property for VisualAge Generator compatibility.

It is recommended that a non-EGL-generated program be identified as externally defined not in the **show** statement, but in the linkage options part that is used at generation time. (The related property is in the linkage options part, `transferToTransaction` element, and is also called **externallyDefined**.) You can make the identification, however, in either way.

#### **Related concepts**

"References to parts" on page 23

#### **Related reference**

"externallyDefined in `transferToTransaction` element" on page 1089

"transferName" on page 1076

## transfer

The EGL **transfer** statement gives control from one main program to another, ends the transferring program, and optionally passes a record whose data is accepted into the receiving program's *input record*. You cannot use a **transfer** statement in a called program.

Your program can transfer control by a statement of the form *transfer to a transaction* or by a statement of the form *transfer to a program*:

- A transfer to a transaction acts as follows--
  - In a main program that runs on IMS/VS or on CICS for z/OS, this statement commits recoverable resources, closes files, closes cursors, and starts a new transaction.
  - In a program that runs as a z/OS main batch program, as an IMS BMP, or as a Java main text or main batch program, the statement starts a program in the same run unit, but pre-transfer behavior depends on the setting of build descriptor option **synchOnTrxTransfer**--
    - If the value of **synchOnTrxTransfer** is NO (the default), the transfer statement does not close or commit resources, which are available to the invoked program.
    - If the value of **synchOnTrxTransfer** is YES, the transfer statement commits recoverable resources, closes files, and closes cursors.
  - In a program of type VGWebTransaction, a transfer to a transaction is not valid; instead, use one of the following statements:
    - A **transfer** statement of the form *transfer to a program* (for transferring to another VGWebTransaction program or to a main batch program); or
    - A **forward** statement (for forwarding control to a URL).
  - In a PageHandler, a transfer to a transaction is not valid; use the **forward** statement instead.
- A transfer to a program starts a program in the same run unit. With exceptions that are described later and that relate to DL/I processing, a transfer to a program does not cause a *synchronization point*: does not close files or cursors and does not commit or rollback recoverable resources.

A transfer to a program is available in programs of type VGWebTransaction (on any system) and in any main COBOL program:

- On CICS for z/OS, the transfer does not cause a synchronization point unless a PSB is scheduled when the transfer occurs and when one of the following situations is in effect:
  - The receiving program begins by scheduling a different PSB or no PSB at all; or
  - The receiving program begins by scheduling the same PSB, but the build descriptor option **synchOnPgmTransfer** is set to YES, which is the default value.

EGL implements the statement with the CICS XCTL command and uses the COMMAREA option of that command to pass the record, after which the record data starts in the first byte of the CICS common area.

- For z/OS batch and IMS BMP programs, a transfer to a program is valid. If the receiving program was generated neither by EGL nor by VisualAge Generator, EGL uses the OS XCTL macro to implement the transfer. In any case, it is not valid to transfer between a z/OS batch program and an IMS BMP program.
- For IMS/VS programs, a transfer to a program is supported only if the receiving program was generated by EGL or VisualAge Generator.

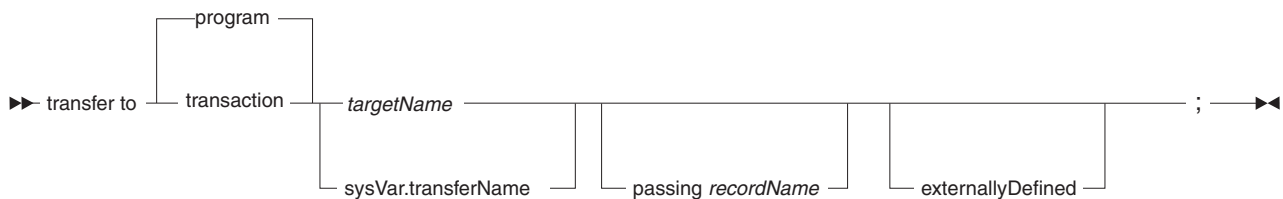
If the initial program in a transaction is a main batch program, transfer to a main transaction program is not supported. If the initial program is a main transaction, transfer is not valid to a main batch program that accesses the I/O PCB.

The linkage options part, **transferLink** element has no effect when you are transferring control from Java code to Java code, but is meaningful otherwise.

If you are transferring control code to code that was *not* written with EGL or VisualAge Generator, it is recommended that you set the linkage options part, **transferLink** element as follows:

- If you are transferring to a transaction, set the **externallyDefined** property to **yes**.
- If you are transferring to a program, set the **linkType** property to *externallyDefined*

If you are running in VisualAge Generator compatibility mode, you can specify the option **externallyDefined** in the transfer statement, as occurs for programs migrated from VisualAge Generator; but it is recommended that you set the equivalent value in the linkage options part instead. For details on VisualAge Generator compatibility mode, see *Compatibility with VisualAge Generator*.



**program** *targetName* (the default)

The program that receives control. If you are generating for COBOL and specify a program name of more than 8 characters, the program name is truncated to 8 characters with character substitutions (if needed), as described in *Name aliasing*.

**transaction** *targetName*

The transaction or program that receives control, as described earlier.

**sysVar.transferName**

A system function that contains a target name that can be set at run time. For details, see *sysVar.transferName*.

**passing** *recordName*

A record that is received as the input record in the target program. The passed record may be of any type, but the length and primitive types must be compatible with the record that receives the data. The input record in the target program must be of type *basicRecord*.

**externallyDefined**

Not recommended for new development, as described earlier.

**Related concepts**

“Compatibility with VisualAge Generator” on page 532

“Name aliasing” on page 774

**Related reference**

“transferName” on page 1076

## try

The EGL **try** statement indicates that the program continues running if a statement of any of the following kinds results in an error and is within the **try** statement:

- An input/output (I/O) statement

- A system-function invocation
- A **call** statement

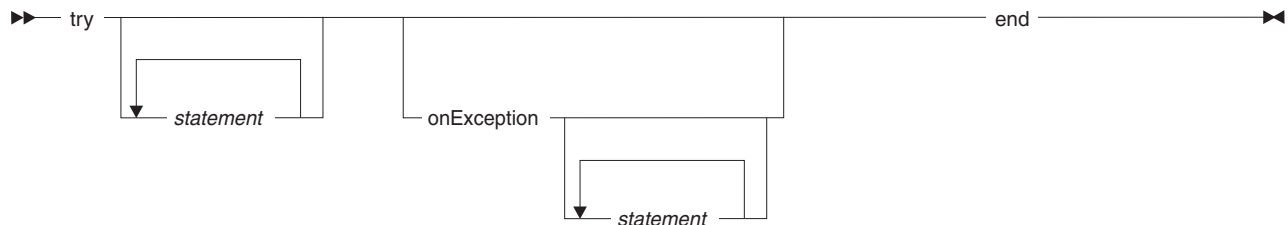
If an exception occurs, processing resumes at the first statement in the **onException** block (if any), or at the first statement following the end of the **try** statement. A hard I/O error, however, is handled only in the following cases:

- If the system variable **VGVar.handleHardIOErrors** is set to 1 and any hard I/O error occurs; or
- If the system variable **DLIVar.handleHardDLIErrors** is set 1 and the hard I/O error occurs during access of a DL/I database or IMS message queue.

Otherwise, the program displays a message (if possible) and ends.

A **try** statement has no effect on runtime behavior when an exception occurs in a function or program that is invoked from within the **try** statement.

For other details, see *Exception handling*.



*statement*

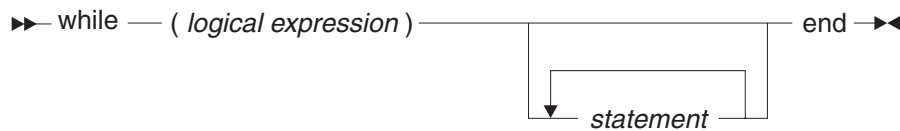
Any EGL statement.

#### **OnException**

A block of statements that run if an exception condition occurs.

## **while**

The EGL keyword **while** marks the start of a set of statements that run in a loop. The first run occurs only if a logical expression resolves to true, and each subsequent iteration depends on the same test. The keyword **end** marks the close of the **while** statement.



*logical expression*

An expression (a series of operands and operators) that evaluates to true or false

*statement*

A statement in the EGL language

An example is as follows:

```

sum = 0;
i = 1;
while (i < 4)
    sum = inputArray[i] + sum;
    i = i + 1;
end

```

#### Related tasks

“Syntax diagram for EGL statements and commands” on page 884

#### Related reference

“Logical expressions” on page 593

“EGL statements” on page 88

---

## Library (generated output)

A library part for Java output is generated as a Java class. The name of the class is the part alias (or is the part name, if no alias is specified), but EGL makes character substitutions as described in *How Java names are aliased*.

#### Related concepts

“Library part of type basicLibrary” on page 169

“Library part of type basicLibrary” on page 169

“Run unit” on page 866

#### Related tasks

“How Java names are aliased” on page 776

#### Related reference

“Library part in EGL source format”

---

## Library part in EGL source format

You declare a library part in an EGL source file, which is described in *EGL source format*.

An example of a library part is as follows:

```

Library CustomerLib3

```

```

// Use declarations
Use StatusLib;

```

```

// Data Declarations
exceptionId ExceptionId ;

```

```

// Retrieve one customer for an email
// In: customer, with emailAddress set
// Out: customer, status

```

```

Function getCustomerByEmail ( customer CustomerForEmail, status int )
    status = StatusLib.success;
    try
        get customer ;
    onException
        exceptionId = "getCustomerByEmail" ;
        status = sqlCode ;
    end
    commit();
end

```

```

// Retrieve one customer for a customer ID
//   In: customer, with customer ID set
//   Out: customer, status
Function getCustomerByCustomerId ( customer Customer, status int )
status = StatusLib.success;
try
  get customer ;
onException
  exceptionId = "getCustomerByCusomerId" ;
  status = sqlCode ;
end
commit();
end

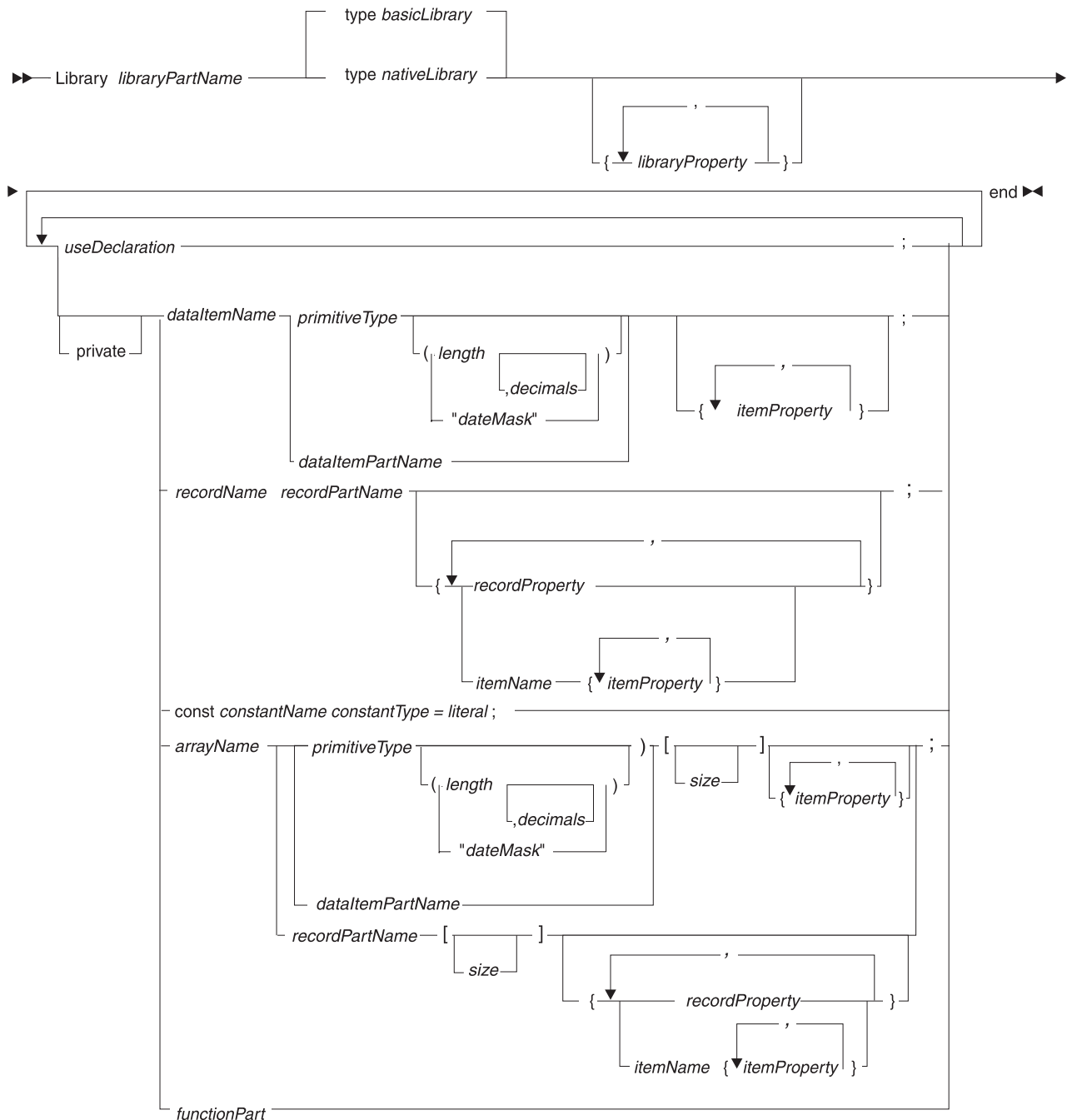
// Retrieve multiple customers for an email
//   In: startId
//   Out: customers, status
Function getCustomersByCustomerId
( startId CustomerId, customers Customer[], status int )
status = StatusLib.success;
try
  get customers usingKeys startId ;
onException
  exceptionId = "getCustomerForEmail" ;
  status = sqlCode ;
end
commit();
end

end

```

The diagram of a library part is as follows:





### **Library** *libraryPartName* ... **end**

Identifies the part as a library part and specifies the name. If you do not set the **alias** property (as described later), the name of the generated library is *libraryPartName*.

For other rules, see *Naming conventions*.

### **type** *BasicLibrary*, **type** *NativeLibrary*, **type** *ServiceBindingLibrary*

Indicates the library type:

- A basic library (type *BasicLibrary*) contains EGL-written functions and values for runtime use in other EGL logic; for details, see *Library part of type BasicLibrary*.

- A native library (type `NativeLibrary`) acts as an interface for an external DLL; for details, see *Library part of type NativeLibrary*
- A service binding library (type `ServiceBindingLibrary`) indicates how a variable of type `Interface` or `Service` accesses a service at run time; for details, see *Library part of type ServiceBindingLibrary*

The library is of type `BasicLibrary` by default.

#### *libraryProperties*

The library properties are as follows:

- **alias**
- **allowUnqualifiedItemReferences**
- **callingConvention** (which is available only in libraries of type `NativeLibrary`)
- **dllName** (which is available only in libraries of type `NativeLibrary`)
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **localSQLScope**
- **messageTablePrefix**
- **runtimeBind** (which is available only in libraries of type `ServiceBindingLibrary`)
- **throwNrfEofExceptions**

All are optional:

- **alias** = "*alias*" identifies a string that is incorporated into the names of generated output. If you do not set the **alias** property, the program-part name is used instead.
- **allowUnqualifiedItemReferences** = **no**, **allowUnqualifiedItemReferences** = **yes** specifies whether to allow your code to reference structure items but to exclude the name of the *container*, which is the data table, record, or form that holds the structure item. Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to `myItem01`, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

- As used in a library of type `NativeLibrary`, **callingConvention** = **I4GL** specifies how the EGL runtime passes data between two kinds of code:

- The EGL code that invokes the library function; and
- The function in the DLL being accessed.

The only value now available for **callingConvention** is *I4GL*. For additional details, see *Library part of type NativeLibrary*.

- As used in a library of type *NativeLibrary*, **dllName** specifies the DLL name, which is final; it cannot be overridden at deployment time. If you do not specify a value for the library property **dllName**, you must specify the DLL name in the Java runtime property `vgj.defaultI4GLNativeLibrary`.

For additional details, see *Library part of type NativeLibrary*.

- **handleHardIOErrors = yes, handleHardIOErrors = no** sets the default value for the system variable `VGVar.handleHardIOErrors`. The variable controls whether a program continues to run after a hard error has occurred on an I/O operation in a try block. The default value for the property is *yes*, which sets the variable to 1.

For other details, see `VGVar.handleHardIOErrors` and *Exception handling*.

- **includeReferencedFunctions = no, includeReferencedFunctions = yes** indicates whether the library contains a copy of each function that is neither inside the library nor in a library accessed by the current library. The default value is *no*, which means that you can ignore this property if all functions that are to be part of this library are inside the library.

If the library is using shared functions that are not in the library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.

- **localSQLScope = yes, localSQLScope = no** indicates whether identifiers for SQL result sets and prepared statements are local to the library code during invocation by a program or pageHandler, as is the default. If you accept the value *yes*, different programs can use the same identifiers independently, and the program or pageHandler that uses the library can independently use the same identifiers as are used in the library.

If you specify *no*, the identifiers are shared throughout the run unit. The identifiers created when the SQL statements in the library is invoked are available in other code that invokes the library, although the other code can use **localSQLScope = yes** to block access to those identifiers. Also, the library may reference identifiers created in the invoking program or pageHandler, but only if the SQL-related statements were already run in the other code and if the other code did not block access.

The effects of sharing SQL identifiers are as follows:

- You can open a result set in one code and get rows from that set in another
- You can prepare an SQL statement in one code and run that statement in another

In any case, the identifiers available when the program or pageHandler accesses the library are available when the same program or pageHandler accesses the same or another function in the same library.

- **msgTablePrefix = "prefix"** specifies the first one to the four characters in the name of a data table that is used as a message table. (The message table is available to forms that are output by library functions.) The other characters in the name correspond to one of the national language codes listed in *DataTable part in EGL source format*.
- **runtimeBind = no, runtimeBind = yes** is valid only in library parts of type *ServiceBindingLibrary* and indicates whether the service binding information

is saved to a library-specific property file that can be changed at deployment time and is accessed only at run time. For details, see *Library part of type ServiceBindingLibrary*.

- **throwNrfEofExceptions = no**, **throwNrfEofExceptions = yes** specifies whether a soft error causes an exception to be thrown. The default is *no*. For background information, see *Exception handling*.

*useDeclaration*

Provides easier access to a data table or library, and is needed to access forms in a form group. For details, see *Use declaration*.

**private**

Indicates that the variable, constant, or function is unavailable outside the library. If you omit the term **private**, the variable, constant, or function is available.

You cannot specify **private** for a function in a library of type NativeLibrary.

*fieldName*

Name of a primitive field. For the rules of naming, see *Naming conventions*.

*primitiveType*

The primitive type of a field or (in relation to an array) the primitive type of an array element.

*length*

The parameter's length or (in relation to an array) the length of an array element. The length is an integer that represents the number of characters or digits in the memory area referenced either by *fieldName* or (in the case of an array) *dynamicArrayName*.

*decimals*

For a numeric type, you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

*"dateTimeMask"*

For TIMESTAMP and INTERVAL types, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the datetime value. The mask is present with the data at run time.

*dataItemPartName*

The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

*recordName*

Name of a record. For the rules of naming, see *Naming conventions*.

*recordPartName*

Name of a record part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

*constantName literal*

Name and value of a constant. The value is either a quoted string or a number. For the rules of naming, see *Naming conventions*.

*itemProperty*

An item-specific property-and-value pair, as described in *Overview of EGL properties and overrides*.

*recordProperty*

A record-specific property-and-value pair. For details on the available properties, see the reference topic for the record type of interest.

A basic record has no properties.

*itemName*

Name of a record field whose properties you wish to override. See *Overview of EGL properties and overrides*.

*arrayName*

Name of a dynamic array. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

*size*

Number of elements in the array. If you specify the number of elements, the array is static; otherwise, the array is dynamic.

*functionPart*

A function. No parameter in the function can be of a loose type. For details, see *Function part in EGL source format*.

**Related concepts**

"EGL projects, packages, and files" on page 15  
"Library part of type basicLibrary" on page 169  
"Library part of type basicLibrary" on page 169  
"Library part of type ServiceBindingLibrary" on page 172  
"Overview of EGL properties" on page 64  
"References to parts" on page 23  
"References to variables in EGL" on page 59  
"Typedef" on page 28

**Related reference**

"Basic record part in EGL source format" on page 461  
"DataTable part in EGL source format" on page 568  
"EGL source format" on page 586  
"Exception handling" on page 94  
"Function part in EGL source format" on page 621  
"Indexed record part in EGL source format" on page 632  
"Input form" on page 859  
"Input record" on page 859  
"I/O error values" on page 638  
"Java runtime properties (details)" on page 642  
"MQ record part in EGL source format" on page 769  
"Primitive types" on page 34  
"Relative record part in EGL source format" on page 865  
"Serial record part in EGL source format" on page 868  
"SQL record part in EGL source format" on page 877  
"VGUIRecord part in EGL source format" on page 1089  
"Use declaration" on page 1091  
"handleHardIOErrors" on page 1082

---

## like operator

In a logical expression, you can compare a text expression against another string (called a *like criterion*), character position by character position from left to right. Use of this feature is similar to use of the SQL keyword **like** in SQL queries.

An example is as follows:

```
// variable myVar01 is the string expression
// whose contents will be compared to a like criterion
myVar01 = "abcdef";

// the next logical expression evaluates to "true"
if (myVar01 like "a_c%")
;
end
```

The like criterion can be either a literal or item of type CHAR or MBCHAR; or an item of type UNICODE. You can include any of these characters in the like criterion:

- % Acts as a wild card, matching zero or more of any characters in the string expression
- \_ (underscore) Acts as a wild card, matching a single character in the string expression
- \ Indicates that the next character is to be compared to a single character in the string expression. The backward virgule (\) is called an *escape character* because it causes an escape from the usual processing; the escape character is not compared to any character in the string expression.

The escape character usually precedes a percent sign (%), an underscore (\_), or another backward virgule.

When you use the backward virgule as an escape character (as is the default behavior), a problem arises because EGL uses the same escape character to allow inclusion of a quote mark in any text expression. In the context of a like criterion, you must specify two backward virgules because the text available at run time is the text that lacks the initial virgule.

It is recommended that you avoid this problem. Specify another character as the escape character by using the escape clause, as shown in a later example. However, you cannot use a double quote mark (") as an escape character.

Any other character in *likeCriterion* is a literal that is compared to a single character in *string expression*.

The following example shows use of an escape clause:

```
// variable myVar01 is the string expression
// whose contents will be compared to a like criterion
myVar01 = "ab%def";

// the next logical expression evaluates to "true"
if (myVar01 like "ab\\%def")
;
end

// the next logical expression evaluates to "true"
if (myVar01 like "ab+%def" escape "+")
;
end
```

The LIKE operator ignores trailing blanks in both operands:

```
// is true
if ("hello " LIKE "hello ")
;
end
```

#### Related reference

“EGL statements” on page 88

“Logical expressions” on page 593

“Text expressions” on page 601

---

## Linkage properties file (details)

When you generate a calling Java program or wrapper, you can specify that linkage information is required at run time. You make that specification by setting the linkage-option values for the called program as follows:

- The value of the callLink element property **type** is `remoteCall` or `ejbCall`; and
- The value of the callLink element property **remoteBind** is `RUNTIME`.

A linkage properties file may be handwritten, but EGL generates a file if (in addition to the settings described earlier) you generate a Java program or wrapper with the build descriptor option **genProperties** set to `GLOBAL` or `PROGRAM`.

### How the linkage properties file is identified at run time

If the callLink element property **remoteBind** for a called program was set to `RUNTIME` in the linkage options part, the linkage properties file is sought at run time; but the source of the file name is different for Java programs and Java wrappers:

- A Java program checks the Java runtime property **cso.linkageOptions.LO**, where *LO* is the name of the linkage options part used for generation. If the property is not present, the EGL runtime code seeks a linkage properties file named **LO.properties**. Again, *LO* is the name of the linkage options part used for generation.

In this case, if the EGL runtime code seeks a linkage properties file but is unable to find that file, an error occurs on the first call statement that requires use of that file. For details on the result, see Exception handling.

- The Java wrapper stores the name of the linkage properties file in the program object variable *callOptions*, which is of type `CSOCallOptions`. The generated name of the file is **LO.properties**, where *LO* is the name of the linkage options part used for generation.

In this case, if the Java Virtual Machine seeks a linkage properties file but is unable to find that file, the program object throws an exception of type `CSOException`.

### Format of the linkage properties file

As used during run time, the linkage properties file includes a series of entries to handle each call from the generated Java program or wrapper that you are deploying.

The primary entry is of type `cso.serverLinkage` and can include any property-and-value pair that you can set in a callLink element of the linkage options part, with the following exceptions:

- Property **remoteBind** is necessarily `RUNTIME` and should not appear



- Property **type** cannot be localCall, because linkage for local calls must be established at generation time

### cso.serverLinkage entries

In the most elementary case, each entry in the linkage properties file is of type `cso.serverLinkage`. The format of the entry is as follows:

`cso.serverLinkage.programName.property=value`

*programName*

The name of the called program. If the called program is generated by EGL, the name you specify is that of a program part.

*property*

Any of the properties appropriate for a Java program, except for properties **remoteBind** and **pgmName**. For details, see *callLink element*.

*value*

A value that is valid for the specified property.

An example for called program Xyz is as follows, where *xxx* refers to a case-sensitive string:

```
cso.serverLinkage.Xyz.type=ejbCall
cso.serverLinkage.Xyz.remoteComType=TCPIP
cso.serverLinkage.Xyz.remotePgmType=EGL
cso.serverLinkage.Xyz.externalName=xxx
cso.serverLinkage.Xyz.package=xxx
cso.serverLinkage.Xyz.conversionTable=xxx
cso.serverLinkage.Xyz.location=xxx
cso.serverLinkage.Xyz.serverID=xxx
cso.serverLinkage.Xyz.parmForm=COMMDATA
cso.serverLinkage.Xyz.providerURL=xxx
cso.serverLinkage.Xyz.luwControl=CLIENT
```

The literal values TCPIP, EGL, and so on are not case sensitive and are examples of valid data.

### cso.application entries

If you wish to create a series of `cso.serverLinkage` entries that refer to any of several called programs, precede those entries with one or more entries of type `cso.application`. Your purpose in this case is to equate a single application name to multiple program names. In the subsequent `cso.serverLinkage` entries, you use the application name instead of *programName*; then, at Java run time, those `cso.serverLinkage` entries handle calls to any of several programs.

The format of a `cso.application` entry is as follows:

`cso.application.wildProgramName.appName`

*wildProgramName*

A valid program name, an asterisk, or the beginning of a valid program name followed by an asterisk. The asterisk is the wild-card equivalent of one or more characters and provides a way to identify a set of names.

If *wildProgramName* refers to a program that is generated by EGL, any program name included in *wildProgramName* is the name of a program part.

*appName*

A series of characters that conforms to the EGL naming conventions. The value of *appName* is used in subsequent `cso.serverLinkage` entries.



The following example show use of an asterisk as a wild-card character. The `cso.serverLinkage` entries in this example handle any call to a program whose name begins with `XYZ`:

```
cso.application.Xyz*=myApp
cso.serverLinkage.myApp.type=remoteCall
cso.serverLinkage.myApp.remoteComType=TCPIP
cso.serverLinkage.myApp.remotePgmType=EGL
cso.serverLinkage.myApp.externalName=xxx
cso.serverLinkage.myApp.package=xxx
cso.serverLinkage.myApp.conversionTable=xxx
cso.serverLinkage.myApp.location=xxx
cso.serverLinkage.myApp.serverID=xxx
cso.serverLinkage.myApp.parmForm=COMMDATA
cso.serverLinkage.myApp.luwControl=CLIENT
```

The following example shows use of the same `cso.serverLinkage` entries to handle calls to any of several programs, even though the names of those programs do not begin with the same characters:

```
cso.application.Abc=myApp
cso.application.Def=myApp
cso.application.Xyz=myApp
cso.serverLinkage.myApp.type=remoteCall
cso.serverLinkage.myApp.remoteComType=TCPIP
cso.serverLinkage.myApp.remotePgmType=EGL
cso.serverLinkage.myApp.externalName=xxx
cso.serverLinkage.myApp.package=xxx
cso.serverLinkage.myApp.conversionTable=xxx
cso.serverLinkage.myApp.location=xxx
cso.serverLinkage.myApp.serverID=xxx
cso.serverLinkage.myApp.parmForm=COMMDATA
cso.serverLinkage.myApp.luwControl=CLIENT
```

If multiple `cso.application` entries are valid for a program, EGL uses the first entry that applies.

#### **Related concepts**

“Linkage options part” on page 399

“Linkage properties file” on page 447

#### **Related tasks**

“Editing the `callLink` element of a linkage options part” on page 401

“Setting up the J2EE runtime environment for EGL-generated code” on page 437

#### **Related reference**

“`callLink` element” on page 499

“Exception handling” on page 94

“Java runtime properties (details)” on page 642

“Naming conventions” on page 778

---

## **matches operator**

In a logical expression, you can compare a string expression against another string (called a *match criterion*), character position by character position from left to right. Use of this feature is similar to use of *regular expressions* in UNIX or Perl.

An example is as follows:

```
// variable myVar01 is the string expression
// whose contents will be compared to a match criterion
myVar01 = "abcdef";

// the next logical expression evaluates to "true"
if (myVar01 matches "a?c*")
;
end
```

The match criterion can be either a literal or item of type CHAR or MBCHAR; or an item of type UNICODE. You can include any of these characters in the match criterion:

- \* Acts as a wild card, matching zero or more of any characters in the string expression
- ? Acts as a wild card, matching a single character in the string expression
- [ ] Acts as a delimiter such that any one of the characters between the two brackets is valid as a match for the next character in the string expression. The following component of a match criterion, for example, indicates that a, b, or c is valid as a match:

[abc]

- Creates a range within the bracket delimiters, such that any character within the range is valid as a match for the next character in the string expression. The following component of a match criterion, for example, indicates that a, b, or c is valid as a match:

[a-c]

The hyphen (-) has no special meaning outside of bracket delimiters.

- ^ Creates a wild-card rule such that, if the caret (^) is the first character inside bracket delimiters, any character other than the delimited characters is valid as a match for the next character in the string expression. The following component of a match criterion, for example, indicates that any character other than a, b, or c is valid as a match:

[^abc]

The caret has no special meaning in these cases:

- It is outside of bracket delimiters
- It is inside of bracket delimiters, but not in the first position

- \ Indicates that the next character is to be compared to a single character in the string expression. The backward virgule (\) is called an *escape character* because it causes an escape from the usual processing; the escape character is not compared to any character in the string expression.

The escape character usually precedes a character that is otherwise meaningful in the match criterion; for example, an asterisk (\*) or a question mark (?).

When you use the backward virgule as an escape character (as is the default behavior), a problem arises because EGL uses the same escape character to allow inclusion of a quote mark in any text expression. In the context of a match criterion, you must specify two backward virgules because the text available at run time is the text that lacks the initial virgule.

It is recommended that you avoid this problem. Specify another character as the escape character by using the escape clause, as shown in a later example. However, you cannot use a double quote mark (") as an escape character.

Any other character in *matchCriterion* is a literal that is compared to a single character in *string expression*.

The following example shows use of an escape clause:

```
// variable myVar01 is the string expression
// whose contents will be compared to a match criterion
myVar01 = "ab*def";

// the next logical expression evaluates to "true"
if (myVar01 matches "ab\\*[abcd][abcde][^a-e]")
;
end

// the next logical expression evaluates to "true"
if (myVar01 matches "ab+*def" escape "+")
;
end
```

#### Related reference

"EGL statements" on page 88

"Logical expressions" on page 593

"Text expressions" on page 601

---

## Message customization for EGL Java run time

When an error occurs at Java run time, an EGL system message is displayed by default; but you can specify a customized message for each of those system messages or for a subset.

When a message is required, EGL first searches a properties file that you identify in the Java runtime property `vgj.messages.file`. The format of the referenced file is the same as for any Java properties file, as described in *Program properties file* and as shown in the current topic.

In many cases, a system message includes placeholders for the message inserts that EGL retrieves at run time. If your code submits an invalid date mask to a system function, for example, the message has two placeholders; one (placeholder 0) for the date mask itself, the other (placeholder 1) for the name of the system function. In properties-file format, the entry for the default message is as follows:

```
VGJ0216E = {0} is not a valid date mask for {1}.
```

You can change the wording of the message to include all or some of the placeholders in any order, but you cannot add placeholders. Valid examples are as follows:

```
VGJ0216E = Function {1} was given invalid date mask {0}.
```

```
VGJ0216E = Function {1} was given an invalid date mask.
```

A fatal error occurs if the file identified in property `vgj.messages.file` cannot be opened.

For details on the message numbers and their meaning, see *EGL Java runtime error codes*.

Other details are available in Java language documentation:

- For details on how messages are processed and on what content is valid, see the documentation for Java class `java.text.MessageFormat`.
- For details on handling characters that cannot be directly represented in the ISO 8859-1 character encoding (which is always used in properties files), see the documentation for Java class `java.util.Properties`.

#### Related concepts

"Program properties file" on page 433

#### Related reference

"EGL Java runtime error codes" on page 1097

"Java runtime properties (details)" on page 642

---

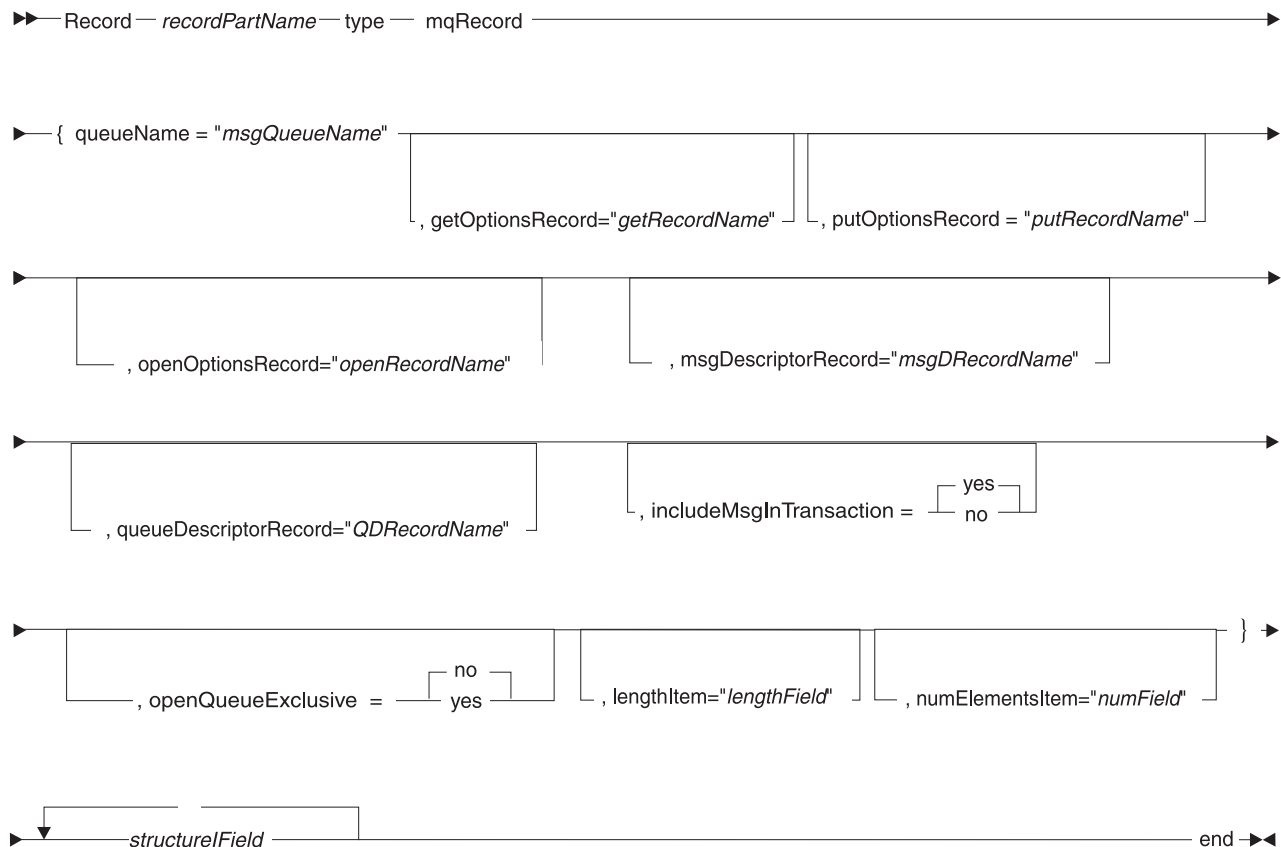
## MQ record part in EGL source format

You can declare MQ record parts in an EGL source file. For an overview of that file, see *EGL source format*. For an overview of how EGL interacts with MQSeries, see *MQSeries support*.

An example of an MQ record part is as follows:

```
Record myMQRecordPart type mqRecord
{
    queueName = "myQueue"
}
10 myField01 CHAR(2);
10 myField02 CHAR(78);
end
```

The syntax diagram for an MQ record part is as follows:



#### Record *recordPartName* **mqRecord**

Identifies the part as being of type **mqRecord** and specifies the name. For rules, see *Naming conventions*.

#### **queueName** = "*msgQueueName*"

The message queue name, which is the logical queue name and usually not the name of the physical queue. For details on the format of your input, see *MQ record properties*.

#### **getOptionsRecord** = "*getRecordName*"

Identifies a program variable (a basic record) that is used as a get options record. For details, see *Options records for MQ records*. This property was formerly the **getOptions** property.

#### **putOptionsRecord** = "*putRecordName*"

Identifies a program variable (a basic record) that is used as a put options record. For details, see *Options records for MQ records*. This property was formerly the **putOptions** property.

#### **openOptionsRecord** = "*openRecordName*"

Identifies a program variable (a basic record) that is used as an open options record. For details, see *Options records for MQ records*. This property was formerly the **openOptions** property.

#### **msgDescriptorRecord** = "*msgDRecordName*"

Identifies a program variable (a basic record) that is used as a message descriptor. For details, see *Options records for MQ records*. This property was formerly the **msgDescriptor** property.

**queueDescriptorRecord = "QDRecordName"**

Identifies a program variable (a basic record) that is used as a queue descriptor. For details, see *Options records for MQ records*. This property was formerly the **queueDescriptor** property.

**includeMsgInTransaction = yes, includeMsgInTransaction = no**

If this property is set to *yes* (the default), each of the record-specific messages is embedded in a transaction, and your code can commit or roll back that transaction. For details on the implications of your choice, see *MQSeries support*.

**openQueueExclusive = no, openQueueExclusive = yes**

If this property is set to *yes*, your code has the exclusive ability to read from the message queue; otherwise, other programs can read from the queue. The default is *no*. This property is equivalent to the MQSeries option `MQOO_INPUT_EXCLUSIVE`.

**lengthItem = "lengthField"**

The length field, as described in *MQ record properties*.

**numElementsItem = "numElementsField"**

The number of elements field, as described in *MQ record properties*.

**structureField**

A structure field, as described in *Structure item in EGL source format*.

### Related concepts

"EGL projects, packages, and files" on page 15

"References to parts" on page 23

"MQSeries support" on page 336

"Parts" on page 19

"Record parts" on page 135

"References to variables in EGL" on page 59

"Typedef" on page 28

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

### Related reference

"Arrays" on page 75

"DataItem part in EGL source format" on page 566

"EGL source format" on page 586

"Function part in EGL source format" on page 621

"Indexed record part in EGL source format" on page 632

"MQ record properties" on page 772

"Naming conventions" on page 778

"Options records for MQ records" on page 772

"Primitive types" on page 34

"Program part in EGL source format" on page 841

"Relative record part in EGL source format" on page 865

"Serial record part in EGL source format" on page 868

"SQL record part in EGL source format" on page 877

"Structure field in EGL source format" on page 880

---

## MQ record properties

This page describes these MQ record properties:

- Queue name
- Include message in transaction
- Open input queue for exclusive use

For details on the other properties, see these pages:

- Options records for MQ records
- Properties that support variable-length records

### Queue name

*Queue name* is required and refers to the logical queue name, which can be no more than 8 characters. For details on the meaning of your input, see *MQSeries-related EGL keywords*.

### Include message in transaction

*Include message in transaction*, if set, embeds each of the record-specific messages in a transaction, and your code can commit or roll back that transaction.

For details on the implications of your choice, see *MQSeries support*.

### Open input queue for exclusive use

If you set *Open input queue for exclusive use*, your code has the exclusive ability to read from the message queue; otherwise, other programs can read from the queue. This property is equivalent to the MQSeries option MQOO\_INPUT\_EXCLUSIVE.

#### Related concepts

“MQSeries-related EGL keywords” on page 339

“MQSeries support” on page 336

“Record types and properties” on page 138

#### Related reference

“Options records for MQ records”

“Properties that support variable-length records” on page 860

### Options records for MQ records

Each MQ record is associated with five *options records*, which EGL uses as arguments in the hidden calls to MQSeries:

- Get options record (MQGMO)
- Put options record (MQPMO)
- Open options record (MQOO: a record with one structure item)
- Message descriptor record (MQMD)
- Queue descriptor record (MQOD)

When you specify an options record as a property of an MQ record, you are referring to a variable that uses a working storage record part (like MQOD) as a typedef. The part resides in an EGL file that is provided with the product, as described in *MQSeries support*. Instead of using the record part as is, you can copy it into your own EGL file and customize the part.

If you do not indicate that a given options record is in use, EGL builds a default record and assigns values, as described in the following sections. The default options records are not available, however, when you access MQSeries without using MQ records.

### Get options record

You can create a get options record based on the MQSeries Get Message Options (MQGMO), which is an argument on MQSeries MQGET calls. If you do not declare a get options record, EGL automatically builds a default named MQGMO, and your generated program does the following:

- Initializes the get options record with the values listed at the beginning of *Data initialization*
- Sets OPTIONS to either MQGMO\_SYNCPOINT or MQGMO\_NO\_SYNCPOINT, depending on whether you set the MQ record property *Include message in transaction*

### Put options record

You can create a put options record based on the MQSeries Put Message Options (MQPMO), which is an argument on MQSeries MQPUT calls. If you do not declare a put options record, EGL automatically builds a default named MQPMO, and your generated program does the following:

- Initializes the put options record with the values listed at the beginning of *Data initialization*
- Sets OPTIONS to either MQPMO\_SYNCPOINT or MQPMO\_NO\_SYNCPOINT, depending on whether you set the MQ record property *Include message in transaction*

### Open options record

The content of the open options record determines the value of the Options parameter that is used in calls to the MQSeries command MQOPEN or MQCLOSE. The open options record part (MQOO) is available, but if you do not declare a record based on that part, EGL automatically builds a default named MQOO as follows:

- On an MQOPEN that is invoked because of an EGL add statement, the generated program sets MQOO.OPTIONS to this:  
MQOO\_OUTPUT + MQOO\_FAIL\_IF QUIESCING
- On an MQOPEN that is invoked because of an EGL get next statement, the generated program sets MQOO.OPTIONS to the following when the message queue record property option *Open input queue for exclusive use* is in effect:  
MQOO\_INPUT\_EXCLUSIVE + MQOO\_FAIL\_IF QUIESCING
- On an MQOPEN that is invoked because of an EGL get next statement, the generated program sets MQOO.OPTIONS to the following when the message queue record property option *Open input queue for exclusive use* is not in effect:  
MQOO\_INPUT\_SHARED + MQOO\_FAIL\_IF QUIESCING
- On an MQCLOSE that is invoked because of an EGL close statement, the generated program sets MQOO.OPTIONS to the following:  
MQCO\_NONE

### Message descriptor record

You can create a message descriptor record based on the MQSeries Message Descriptor (MQMD), which is a parameter on MQGET and MQPUT calls. If you do not declare a message descriptor record, EGL automatically builds a default named MQMD and initializes that record with the values listed in *Data initialization*.



## Queue descriptor record

You can create a queue descriptor record based on the MQSeries Object Descriptor (MQOD), which is an argument on MQSeries MQOPEN and MQCLOSE calls. If you do not declare a queue descriptor record, EGL automatically builds a default named MQOD, and your generated program does the following:

- Initializes the queue descriptor record with the values listed at the beginning of *Data initialization*
- Sets OBJECTTYPE in that record to MQOT\_Q
- Sets OBJECTMGRNAME to the queue manager name specified in the system word *record.resourceAssociation*; but if *record.resourceAssociation* does not reference the queue manager name, OBJECTMGRNAME has no value
- Sets OBJECTNAME to the queue name in *record.resourceAssociation*

### Related concepts

“Direct MQSeries calls” on page 341

“MQSeries-related EGL keywords” on page 339

“MQSeries support” on page 336

### Related reference

“Data initialization” on page 564

“recordName.resourceAssociation” on page 985

“MQ record properties” on page 772

---

## Name aliasing

If you use a name that is not valid in Java output, the generator creates and uses an alias for the name in the generated code, for any of these reasons:

- Differences in identifier characters allowed
- Differences in length limitations
- Differences in support for uppercase and lowercase characters
- Using a word that is a reserved word in the generated language
- Using a word that clashes with the name alias syntax (for example, **class\$** is aliased because **class\$** is the alias for **class** in Java generation)

An alias may be generated by substituting a valid set of characters for an invalid character, by truncating names that are too long, by adding a prefix or suffix to a name, or by producing a completely different name such as **EZE00123**.

### Related concepts

#### Related tasks

“Creating an EGL program part” on page 147

#### Related reference

“How Java names are aliased” on page 776

“How Java wrapper names are aliased” on page 776

“Naming conventions” on page 778

## Changes to EGL identifiers in JSP files and generated Java beans

You assign names to PageHandler functions, records, and items in accordance with the rules detailed in *Naming conventions*. However, EGL uses a variation of those

names when creating Java identifiers in JSP files and in the Java bean that is derived from a PageHandler. You need to be aware of those variations if you use the source tab to edit a JSP file, if you use the Properties view, or if you work outside of the EGL-enabled tooling altogether.

The variations are as follows:

- The letters *EGL* precede the names of the PageHandler records, items, and functions. The purpose of this variation is to protect you from errors that could result in the Java runtime environment as a result of differences between the Java bean specification and the naming conventions in EGL.
- In several situations, a suffix is added to the name of a variable that is bound to a particular kind of output control:
  - If you bind an item to a Boolean check box, the Java identifier includes the suffix *AsBoolean*
  - If you bind an item to a selection control (a list box, combo box, radio button group, or check box group) and reference the item in the JavaServer Faces `selectItems` tag, the Java identifier includes the suffix *AsSelectItemsList*
  - If you bind an item to a check box in a JavaServer Faces data table (specifically, if the item is referenced in an `inputRowSelect` tag), the Java identifier includes the suffix *AsIntegerArray*

Aside from the variations listed earlier, EGL attempts to create an identifier that exactly matches the name in the PageHandler.

Consider the PageHandler *myJSP*, which includes variable *myItem*. If you bind that variable to a Boolean check box, the JSP file references the Java-bean property *myJSP.EGLmyItemAsBoolean*, and the Java-bean getter and setter functions are named as follows:

- *getEGLmyItemAsBoolean*
- *setEGLmyItemAsBoolean*

The source for the Boolean check-box tag in the JSP file is as follows:

```
<h:selectBooleanCheckbox styleClass="selectBooleanCheckbox"
    id="checkbox1" value="#{myJSP.EGLmyItemAsBoolean}">
</h:selectBooleanCheckbox>
```

EGL avoids generating a name that would not be valid in Java; for details, see *How Java names are aliased*.

### Related concepts

“PageHandler” on page 223

### Related tasks

“Creating an EGL field and associating it with a Faces JSP” on page 242

“Associating an EGL record with a Faces JSP” on page 243

“Using the Quick Edit view for PageHandler code” on page 244

### Related reference

“How Java names are aliased” on page 776

“Naming conventions” on page 778

“Page Designer support for EGL” on page 221

## How Java names are aliased

When you give a part a name, that name must be a valid Java identifier, except that you can use a hyphen or minus sign (-) in a part name. However, a hyphen cannot be the first character in a part name.

If you choose a name that is a Java keyword or a name that contains a dollar sign (\$) or a hyphen or minus, the part name will not match the name in the generated output. An aliasing mechanism automatically appends a dollar sign to each part name that is a Java keyword. If you specify a name that contains one or more dollar signs or hyphens, the aliasing mechanism replaces each symbol with a Unicode value as follows:

```
$ $0024  
- $002d
```

For example, an item named **class** is aliased to **class\$**, and an item named **class\$** is aliased to **class\$0024**.

The case you use to declare a part name is preserved. Programs XYZ and xyz are generated in XYZ.java and xyz.java respectively. On Windows 2000/NT/XP, if you generate into the same directory parts with names that differ only in case, the older files are overwritten.

EGL package names are always converted to lower case Java package names.

Finally, if the name of a program, PageHandler, or library matches the name of a class from the Java system package java.lang, a dollar sign is appended to the class name: Object becomes Object\$, Error becomes Error\$, and so on.

For details on how EGL creates Java identifiers in JSP files and in the Java bean that is derived from a PageHandler, see *Changes to EGL identifiers in JSP files and generated Java beans*.

### Related concepts

"Name aliasing" on page 774

### Related reference

"Changes to EGL identifiers in JSP files and generated Java beans" on page 774

## How Java wrapper names are aliased

The EGL generator applies the following rules to alias Java wrapper names:

1. If the EGL name is all uppercase, convert it to lowercase.
2. If the name is a class name or a method name, make the first character uppercase. (For example, the getter method for x is **getX()** not **getx()**.)
3. Delete every underscore (\_) and hyphen (-). (Hyphens are valid in EGL names if you use VisualAge Generator compatibility mode.) If a letter follows the underscore or hyphen, change that character to uppercase.
4. If the name is a qualified name that uses a period (.) as a separator, replace every period with a low line, and add a low line at the beginning of the name.
5. If the name contains a dollar sign (\$), replace the dollar sign with two low lines and add a low line at the beginning of the name.
6. If a name is a Java keyword, add a low line at the beginning of the name.
7. If the name is \* (an asterisk, which represents a filler item), rename the first asterisk **Filler1**, the second asterisk **Filler2**, and so forth.

In addition, special rules apply to Java wrapper class names for program wrappers, record wrappers, and substructured array items. The remaining sections discuss these rules and give an example. In general, if naming conflicts exist between fields within a generated wrapper class, the qualified name is used to determine the class and variable names. If the conflict is still not resolved, an exception is thrown at generation time.

### Program wrapper class

Record parameter wrappers are named by using the above rules applied to the type definition name. If the record wrapper class name conflicts with the program class name or the program wrapper class name, **Record** is added at the end of the record wrapper class name.

The rules for variable names are as follows:

1. The record parameter variable is named using above rules applied to the parameter name. Therefore, the **get()** and **set()** methods contain these names rather than the class name.
2. The **get** and **set** methods are named **get** or **set** followed by the parameter name with the above rules applied.

### Record wrapper class

The rules for substructured array items class names are as follows:

1. The substructured array item becomes an inner class of the record wrapper class, and the class name is derived by applying the above rules to the item name. If this class name conflicts with the containing record class name, **Structure** is appended to the item class name.
2. If any item class names conflict with each other, the qualified item names are used.

The rules for **get** and **set** method names are as follows:

1. The methods are named **get** or **set** followed by the item name with the above rules applied.
2. If any item names conflict with each other, the qualified item names are used.

### Substructured array items class

The rules for substructured array items class names are as follows:

1. The substructured array item becomes an inner class of the wrapper class generated for the containing substructured array item, and the class name is derived by applying the above rules to the item name.
2. If this class name conflicts with the containing substructured array item class name, **Structure** is appended to the item class name.

The rules for **get** and **set** method names are as follows:

1. The methods are named **get** or **set** followed by the item name with the above rules applied.
2. If any item names conflict with each other, the qualified item names are used.

### Example

The following sample program and generated output show what should be expected during wrapper generation:

**Sample program:**

```

Program WrapperAlias(param1 RecordA)

end

Record RecordA type basicRecord
  10 itemA CHAR(10)[1];
  10 item_b CHAR(10)[1];
  10 item$C CHAR(10)[1];
  10 static CHAR(10)[1];
  10 itemC CHAR(20)[1];
  15 item CHAR(10)[1];
  15 itemD CHAR(10)[1];
  10 arrayItem CHAR(20)[5];
  15 innerItem1 CHAR(10)[1];
  15 innerItem2 CHAR(10)[1];
end

```

### Generated output:

Names of generated output

Output	Name
Program wrapper class	<b>WrapperaliasWrapper</b> , containing a field <b>param1</b> , which is an instance of the record wrapper class <b>RecordA</b>
Parameter wrapper classes	<p><b>RecordA</b>, accessible through the following methods:</p> <ul style="list-style-type: none"> <li>• <b>getItemA</b> (from itemA)</li> <li>• <b>getItemB</b> (from the first item-b)</li> <li>• <b>get_Item__C</b> (from item\$C)</li> <li>• <b>get_Static</b> (from static)</li> <li>• <b>get_ItemC_itemB</b> (from itemB in itemC)</li> <li>• <b>getItemD</b> (from itemD)</li> <li>• <b>getArrayItem</b> (from arrayItem)</li> </ul> <p><b>ArrayItem</b> is an inner class of <b>RecordA</b> that contains fields that can be accessed through <b>getInnerItem1</b> and <b>getInnerItem2</b>.</p>

### Related concepts

“Compatibility with VisualAge Generator” on page 532  
 “Java wrapper” on page 390  
 “Name aliasing” on page 774

### Related tasks

“Generating Java wrappers” on page 390

### Related reference

“Java wrapper classes” on page 652  
 “Naming conventions”  
 “Output of Java wrapper generation” on page 782

---

## Naming conventions

This page describes the rules for naming parts and variables and for assigning values to properties such as **file name**. For details on how logic parts can reference areas of memory, see *References to variables and constants* and *Arrays*.

Three categories of identifier are in EGL:

- EGL part and variables names, as described later.

- External resource names that are specified as property values in part or variable declarations. These names represent special cases, and the naming conventions depend on the conventions of the runtime system.
- EGL package names such as com.mycom.mypack. In this case, each character sequence is separated from the next by a period, and each sequence follows the naming convention for an EGL part name. For details on the relationship of package names and file structure, see *EGL projects, packages, and files*.

An EGL part or variable name is a series of 1 to 128 characters. Except as noted, a name must begin with a Unicode letter or underscore and can include additional Unicode letters as well as digits and currency symbols. Other restrictions are in effect:

- The first characters cannot be EZE in any combination of uppercase and lowercase
- A name cannot contain embedded blanks or be an EGL reserved word

Special considerations apply to parts:

- In a record part, the name of a logical file or queue can be no more than 8 characters
- In various parts, the *alias* is incorporated into the names of generated output files and Java classes. If the external name is not specified, the name of the program part is used but is truncated (if necessary) to the maximum number of characters allowed in the runtime environment.

When you generate a program for CICS for z/OS, the name of the output file can be no more than eight characters.

If your code is compatible with VisualAge Generator, the following rule applies to part and variable names but have no effect on package names: Characters after the initial character can include "at" signs (@), hyphens (-), and pound signs (#).

#### Related concepts

"Compatibility with VisualAge Generator" on page 532

"EGL projects, packages, and files" on page 15 "EGL services and Web services" on page 158

"Name aliasing" on page 774

"References to variables in EGL" on page 59

#### Related reference

"Arrays" on page 75

"Changes to EGL identifiers in JSP files and generated Java beans" on page 774

"EGL reserved words" on page 581

"EGL system limits" on page 590

---

## Operators and precedence

The next table lists the EGL operators in order of decreasing precedence. Except for the unary plus (+), minus (-), and not (!), each operator works with two operands.

Operators (separated by commas)	Type of operator	Meaning
+, -	Numeric, unary	Unary plus (+) or minus (-) is a sign before an operand or parenthesized expression, not an operator between two expressions.

Operators (separated by commas)	Type of operator	Meaning
**	Numeric	** is the <i>toThePowerOfInteger</i> operator, which accepts a number to the specified power. For example <code>c = a**b</code> will result in <code>c</code> being assigned the value of $(a^b)$ . The first operand ( <code>a</code> in the example above) cannot have a negative value. The second operand ( <code>b</code> in the example above) must be an integer, or a numeric field with precision 0. The second operand can be positive, negative or 0.
*, /, %	Numeric	Multiplication (*) and integer division (/) are of equal precedence. The division of integers retains a fractional value, if any; for example, <code>7/5</code> yields 1.4.  % is the <i>remainder</i> operator, which resolves to the modulus when the first of two operands or numeric expressions is divided by the second; for example, <code>7%5</code> yields 2.
+, -	Numeric	Addition (+) and subtraction (-) are of equal precedence.
=	Numeric or string	= is the <i>assignment</i> operator, which copies a numeric or character value from an expression or operand into an operand.
!	Logical, unary	! is the <i>not</i> operator, which resolves to a Boolean value (true or false) opposite to the value of a logical expression that immediately follows. That subsequent expression must be in parentheses.
==, !=, <, >, <=, >=, in, is, not	Logical for comparison	The logical operators used for comparison are of equal precedence and are described in the page on logical expressions. Each operator resolves to true or false.
&&	Logical	&& is the <i>and</i> operator, which means "both must be true." The operator resolves to true if the logical expression that precedes the operator is true and if the logical expression that follows the operator is true; otherwise, && resolves to false.
	Logical	is the <i>or</i> operator, which means "one or the other or both." The operator resolves to true if the logical expression that precedes the operator is true or if the logical expression that follows the operator is true or if both are true; otherwise    resolves to false.

You may override the usual precedence (also called the *order of operations*) by using parentheses to separate one expression from another. Operations that have the same precedence in an expression are evaluated in left-to-right order.

#### Related reference

"in operator" on page 629

"Logical expressions" on page 593

“Numeric expressions” on page 600

“Primitive types” on page 34

“Text expressions” on page 601

---

## Output of Java program generation

The output of Java server program generation is as follows:

- A build plan, if the build descriptor option **genProject** is omitted
- Java source code (see *Java program, PageHandler, and library*)
- Related objects needed to prepare and run your program (see *Java program, PageHandler, and library*)
- J2EE environment file
- Program properties file
- A results file, if **genProject** is omitted

You can use the EGL generator to generate entire Java programs. Programs and records are generated as separate Java classes. Functions are generated as methods in the program. Data items and structure items are generated as fields of the record or program class to which they belong.

The following table shows the names of the various types of generated Java parts:

Names of generated Java parts

Part type and name	What is generated
Program named P	A class named P in P.java
Function named F in program P	A method of the P class called \$funcF in P.java
Record named R	A class named EzeR in EzeR.java
Basic record named R, parameter to Function F	A class named Eze\$paramR in Eze\$paramR.java
Linkage options part named L	Linkage properties file named L.properties
Library named Lib	A class named Lib in Lib.java
DataTable named DT	A class named EzeDT in EzeDT.java
Form named F	A class named EzeF in EzeF.java
FormGroup named FG	A class named FG in FG.java

1. For the indicated part types, it is possible that two or more parts may exist with the same name. In that event the name of the second one will have an additional suffix, \$v2. The name of the third will have a \$v3 suffix, the fourth will have \$v4, etc.

If the naming format would cause two names to be identical, EGL adds a suffix to each file generated after the first. The suffix is as follows:

\$vn

where

**n** Is an integer assigned in sequential order, beginning with 2.

### Related concepts

“Build plan” on page 413

“J2EE environment file” on page 440

“Java program, PageHandler, and library” on page 414



“Linkage properties file” on page 447  
“Program properties file” on page 433  
“Results file” on page 414

**Related reference**  
“callLink element” on page 499

---

## Output of Java wrapper generation

The output of Java wrapper generation is as follows:

- A build plan, if the build descriptor option **genProject** is omitted
- JavaBeans™ for wrapping calls to a Java server program (see *Java wrapper*)
- EJB session beans under certain circumstances; for details, see the explanation of the callLink element in *Linkage options part*
- A results file, if **genProject** is omitted

You can use the generated beans to wrap calls to server programs from non-EGL Java classes such as servlets, EJBs, or Java applications. The following types of classes are generated:

- Beans for servers
- Beans for record parameters
- Beans for record array rows

The following table shows the names of the various types of generated Java wrapper parts:

Names of generated Java wrapper parts

Part type and name	What is generated
Program named P	A class named PWrapper in PWrapper.java
Record named R used as a parameter	A class named R in R.java
Substructured area S in record R used as a parameter	A class named R.S in R.java
Linkage options part named L	Linkage properties file named L.properties

1. For the indicated part types, it is possible that two or more parts may exist with the same name. In that event the name of the second one will have an additional suffix, \$v2. The name of the third will have a \$v3 suffix, the fourth will have \$v4, etc.

When you request that a program part be generated as a Java wrapper, EGL produces Java class for each of the following executables:

- The program part
- Each record that is declared as a program parameter
- A session bean, if you specify a linkage options part and a the **callLink** element for the generated program has a link type of **ejbCall**

In addition, the class generated for each record includes an inner class (or a class within an inner class) for each structure item that has these characteristics:

- Is in the internal structure of one of those records
- Has at least one subordinate structure item; in other words, is substructured

- Is an array; in this case, a substructured array

Each generated class is stored in a file. The EGL generator creates names used in Java wrappers as follows:

- The name is converted to lowercase.
- Each hyphen or minus (-) or underscore (\_) is deleted. A character that follows a hyphen or underscore is changed to uppercase.
- When the name is used as a class name or within a method name, the first character is translated back to uppercase.

If one of the parameters to the program is a record, EGL generates a wrapper class for that variable as well. If program Prog has a record parameter with a typeDef named Rec, the wrapper class for the parameter will be called Rec. If the typeDef of a parameter has the same name as the program, the wrapper class for the parameter will have a "Record" suffix.

The generator also produces a wrapper if a record parameter has an array item and the item has other items under it. This substructured array wrapper becomes an inner class of the record wrapper. In most cases, a substructured array item called AItem in Rec will be wrapped by a class called Rec.AItem. The record may contain two substructured array items with the same name, in which case the item wrappers are named by using the qualified names of the items. If the qualified name of the first AItem is Top1.AItem and the qualified name of the second is Top2.Middle2.AItem, the classes will be named Rec.Top1\$\_aItem and Rec.Top2\$\_middle2\$\_aItem. If the name of a substructured array is the same as the name of the program, the wrapper class for substructured array will have a Structure suffix.

Methods to set and get the value of low-level items are generated into each record wrapper and substructured array wrapper. If two low-level items in the record or substructured array have the same name, the generator uses the qualified-name scheme described in the previous paragraph.

Additional methods are generated into wrappers for SQL record. For each item in the record, the generator creates methods to get and set its null indicator value and methods to get and set its SQL length indicator.

You can use the Javadoc tool to build a *classname.html* file once the the class has been compiled. The HTML file describes the public interfaces for the class. If you use HTML files created by Javadoc, be sure that it is an EGL Java wrapper. HTML files generated from a VisualAge Generator Java wrapper are different from those generated from an EGL Java wrapper.

## Example

An example of a record part with a substructured array is as follows:

```
Record myRecord type basicRecord
  10 MyTopStructure[3];
    15 MyStructureItem01 CHAR(3);
    15 MyStructureItem02 CHAR(3);
end
```

In relation to the program part, the output file is named as follows:

*aliasWrapper.java*

where

**alias**

Is the alias name, if any, that is specified in the program part. If the external name is not specified, the name of the program part is used.

In relation to each record declared as a program parameter, the output file is named as follows:

*recordName.java*

where

**recordName**

Is the name of the record part

In relation to a substructured array, the name and position of the inner class depends on whether the array name is unique in the record:

- If the array name is unique in the record, the inner class is within the record class and is named as follows:

*recordName.siName*

where

**recordName**

Is the name of the record part

**siName**

Is the name of the array

- If the array name is not unique in the record, the name of the inner class is based on the fully qualified name of the array, with one qualifier separated from the next by a combination of dollar sign (\$) and underscore (\_). For example, if the array is at the third level of the record, the generated class is an inner class of the record class and is named as follows:

*Topname\$\_Secondname\$\_Siname*

where

**Topname**

Is the name of the top-level structure item

**Secondname**

Is the name of the second-level structure item

**Siname**

Is the name of the substructured-array item

If another, same-named array is immediately subordinate to the highest level of the record, the inner class is also within the record class and is named as follows:

*Topname\$\_Siname*

where

**Topname**

Is the name of the highest-level structure item

**Siname**

Is the name of the substructured-array item

Finally, consider the following case: a substructured array has a name that is not unique in the record, and the array is subordinate to another substructured array whose name is not unique in the record. The class for the subordinate array is generated as an inner class of an inner class.

When you generate a Java wrapper, you also generate a Java properties file and a linkage properties file if you request that linkage options be set at run time.

**Related concepts**

"Build plan" on page 413  
 "Enterprise JavaBean (EJB) session bean" on page 402  
 "Java wrapper" on page 390  
 "Linkage options part" on page 399  
 "Linkage properties file" on page 447  
 "Results file" on page 414

**Related tasks**

"Generating Java wrappers" on page 390

**Related reference**

"callLink element" on page 499  
 "Java wrapper classes" on page 652

---

## PageHandler part in EGL source format

You declare a pageHandler part in an EGL file, which is described in *EGL projects, packages, and files*. This part is a generatable part, which means that it must be at the top level of the file and must have the same name as the file.

An example of a pageHandler part is as follows:

```
// Page designer requires that all pageHandlers
// be in a package named "pagehandlers".
package pagehandlers ;

PageHandler ListCustomers
{onPageLoadFunction="onPageLoad"}

// Library for customer table access
use CustomerLib3;

// List of customers
customerList Customer[] {maxSize=100};

Function onPageLoad()

// Starting key to retrieve customers
startkey CustomerId;

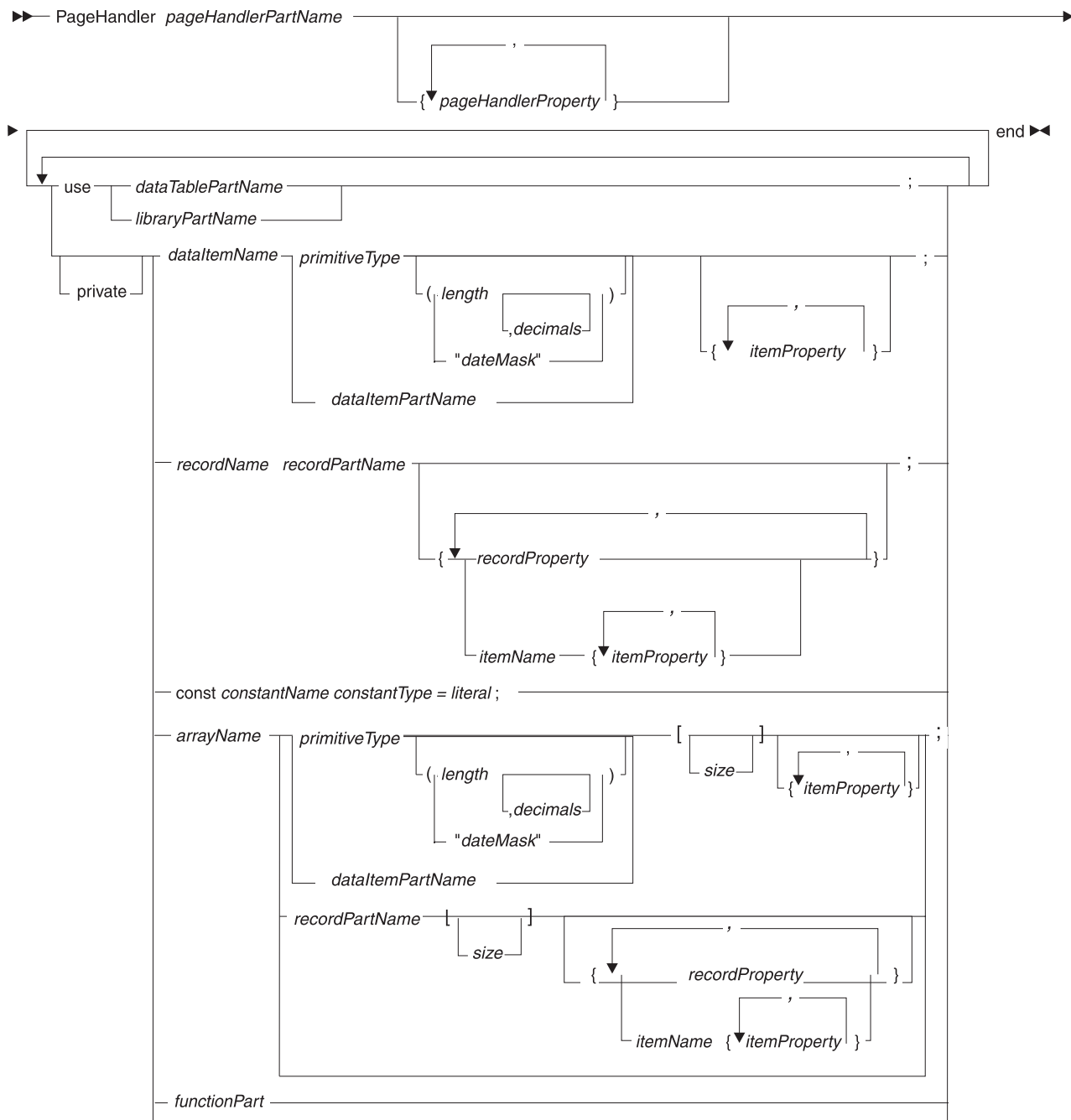
// Result from library call
status int;

// Retrieve up to 100 customer records
startKey = 0;
CustomerLib3.getCustomersByCustomerId(startKey,
customerList, status);

if ( status != 0 && status != 100 )
  setError("Retrieval of Customers Failed.");
end
end

Function returnToIntroductionClicked()
  forward to "Introduction";
end
End
```

The diagram of a pageHandler part is as follows:



### **PageHandler** *pageHandlerPartName* ... **end**

Identifies the part as a PageHandler and specifies the part name. For the rules of naming, see *Naming conventions*.

#### *pageHandlerProperty*

A PageHandler part property, as listed in *PageHandler part properties*.

#### **use** *dataTablePartName*, **use** *libraryPartName*

A use declaration that simplifies access of a data table or library. For details, see *Use declaration*.

#### **private**

Indicates that the variable, constant, or function is unavailable to the JSP that

renders the Web page. If you omit the term **private**, you can bind the variable, constant, or function to a control on the Web page.

*dataItemName*

Name of a data item (a variable). For rules, see *Naming conventions*.

*primitiveType*

The primitive type assigned to the data item.

*length*

The structure item's length, which is an integer. The value of a memory area that is based on the structure item includes the specified number of characters or digits.

*decimals*

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

*dataItemPartName*

The name of a dataItem part that is a model of format for the data item, as described in *typeDef*. The dataItem part must be visible to the pageHandler part, as described in *References to parts*.

*itemProperty*

An item property. For details, see *Page item properties*.

*recordName*

Name of a record (a variable). For rules, see *Naming conventions*.

*recordPartName*

The name of a record part that is a model of format for the record, as described in *typeDef*. The record part must be visible to the pageHandler part, as described in *References to parts*.

*recordProperty*

An override of a record property. For details on the record properties, see one of the following descriptions, depending on the type of record in *recordPartName*:

- Basic record part in EGL source format
- Indexed record part in EGL source format
- MQ record part in EGL source format
- Relative record part in EGL source format
- Serial record part in EGL source format
- SQL record part in EGL source format

*itemName*

Name of the record item whose properties you intend to override.

*itemProperty*

An override of an item property. For details, see *Overview of EGL properties and overrides*.

*constantName literal*

Name and value of a constant. For rules, see *Naming conventions*.

*arrayName*

Name of a dynamic or static array of records or data items. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

*functionPart*

An embedded function. For details on the syntax, see *Function part in EGL source format*.

### Related concepts

"EGL projects, packages, and files" on page 15

"Overview of EGL properties" on page 64

"PageHandler" on page 223

"References to parts" on page 23

"References to variables in EGL" on page 59

"Typedef" on page 28

"viewRootVar property" on page 230

### Related reference

"Exception handling" on page 94

"Function part in EGL source format" on page 621

"Naming conventions" on page 778

"PageHandler field properties" on page 792

"PageHandler part properties"

"Primitive types" on page 34

"Reference compatibility in EGL" on page 862

"setError()" on page 1037

"Use declaration" on page 1091

## PageHandler part properties

Excluding properties that are specific to PageHandler *fields*, the properties of the PageHandler are as follows and are optional:

**alias** = "*alias*"

A string that is incorporated into the names of generated output. If you do not specify an alias, the PageHandler part name is used instead.

**allowUnqualifiedItemReferences** = no, **allowUnqualifiedItemReferences** = yes

Specifies whether to allow your code to reference structure items but to exclude the name of the *container*, which is the data table, record, or form that holds the structure item. Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to myItem01, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

**handleHardIOErrors = yes, handleHardIOErrors = no**

Sets the default value for the system variable **VGVar.handleHardIOErrors**. The variable controls whether a program continues to run after a hard error has occurred on an I/O operation in a try block. The default value for the property is *yes*, which sets the variable to 1.

For other details, see *VGVar.handleHardIOErrors* and *Exception handling*.

**includeReferencedFunctions = no, includeReferencedFunctions = yes**

Indicates whether the PageHandler bean contains a copy of each function that is neither inside the PageHandler nor in a library accessed by the PageHandler. The default value is *no*, which means that you can ignore this property if you are fulfilling the following practices at development time, as is recommended:

- Place shared functions in a library
- Place non-shared functions in the PageHandler

If you are using shared functions that are not in a library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.

**localSQLScope = no, localSQLScope = yes**

Indicates whether identifiers for SQL result sets and prepared statements are local to the pageHandler, as is the default. If you accept the value *yes*, different programs that are called from the pageHandler can use the same identifiers independently.

If you specify *no*, the identifiers are shared throughout the run unit. The identifiers created in the current code are available elsewhere, although other code can use **localSQLScope = yes** to block access to those identifiers. Also, the current code may reference identifiers created elsewhere, but only if the other code was already run and did not block access.

The effects of sharing SQL identifiers are as follows:

- You can open a result set in a pageHandler or called program and get rows from that set in the other code
- You can prepare an SQL statement in one code and run that statement in another

**msgResource = "fileName"**

Identifies a Java resource bundle or properties file that is used in error-message presentation. The Java runtime searches for the file, which must be in a directory listed in the Java classpath.

The search is for a file whose name begins with *fileName*, with an environment-specific locale value appended to that name. The Norwegian version of a resource bundle called *myFile*, for example, is *myFile\_no*.

First, the runtime search is for a file with extension *class* (for a resource bundle); but if that file is not found, the search is for a file with extension *properties* (for a properties file).

The content of the resource bundle or properties file is composed of a set of keys and related values. A particular value is displayed in response to the



program's invoking the EGL system function `sysLib.setError`, when the invocation includes use of the key for that value.

**onPageLoadFunction = "functionName"**

The name of a PageHandler function that receives control when the related JSP initially displays a Web page. This function can be used to set up initial values of the data displayed in the page. This property was formerly the **onPageLoad** property.

Arguments passed to the function must be reference-compatible, as described in *Reference Compatibility in EGL*.

**scope = session, scope = request**

Specifies when the PageHandler (the page bean) is removed from memory in the Web application server:

**If scope is set to *session* (as is the default)**

After the Web page is presented, the PageHandler remains in memory; then, if the user invokes a PageHandler function, the values returned from the Web page are available to the function--

- If the function ends without issuing a **forward**, the JSF runtime code re-displays the same Web page without re-invoking the OnPageLoad function.
- If the function ends with a **forward** statement that refers to the same page, the situation is the same as in the previous case; *the OnPageLoad function is not re-invoked*
- If the function ends with a **forward** statement that refers to a different page, the data in the PageHandler is lost, although you may pass arguments and retain values in either the session or request object; for details, see *forward*

Even when the PageHandler **scope** property is set to *session*, the data in that PageHandler is not necessarily available throughout the user session. If PageHandler A forwards control to PageHandler B, for example, the data in PageHandler A is no longer available unless you took action like the following:

- Included the data in arguments that are passed to the new PageHandler; or
- Included the data in the session or request object; or
- Included the data in a relational database or other data store.

**If scope is set to *request***

After the Web page is presented, the PageHandler remains in memory, but is removed from memory as soon as the user submits another request. When the user invokes a PageHandler function, the following events occur:

- The previous PageHandler is removed from memory
- The requested PageHandler is started (or restarted), and the OnPageLoad function is invoked
- The data received from the user is assigned to the related variables in the PageHandler
- The PageHandler function runs, and the immediate events are the same as when the scope is set to *session*--
  - If the function ends without issuing a **forward** statement, the JSF runtime re-displays the same Web page without re-invoking the OnPageLoad function.

- If the function ends with a **forward** statement that refers to the same page, the situation is the same as in the previous case; *the OnPageLoad function is not re-invoked*
- If the function ends with a **forward** statement that refers to a different page, the data in the PageHandler is lost, although you may pass arguments and retain values in either the session or request object; for details, see *forward*

It is recommended that you set this property explicitly to document your decision, which affects the design and operation of your Web application.

**throwNrfEofExceptions = no, throwNrfEofExceptions = yes**

Specifies whether a soft error causes an exception to be thrown. The default is *no*. For background information, see *Exception handling*.

**title = "literal"**

The **title** property is a bind property, which means that the assigned value is used as a default when you are working in Page Designer. The property specifies the title of the page.

*literal* is a quoted string.

**validationBypassFunctions = ["functionNames"]**

Identifies one or more *event handlers*, which are PageHandler functions that are associated with a button control in the JSP. Each function name is separated from the next by a comma.

If you specify an event handler in this context, the EGL runtime skips input-field and page validations when the user clicks the button or hypertext link that is or related to the event handler. This property is useful for reserving a user action that ends the current PageHandler processing and that immediately transfers control to another Web resource.

**validatorFunction = "functionName"**

Identifies the PageHandler validator function, which is invoked after all the field validators are invoked, as described in *Validation in Web applications built with EGL*. This property was formerly the **validator** property.

**view = "JSPFileName"**

Identifies the name and subdirectory path to the Java Server Page (JSP) that is bound to the PageHandler. *JSPFileName* is a quoted string.

The default value is the name of the PageHandler, with the file extension **.jsp**. If you specify this property, include the file extension, if any.

When you save or generate a PageHandler, EGL adds a JSP file to your project for subsequent customization, unless a JSP file of the same name (the name specified in the **view** property) is already in the appropriate folder (the folder WebContent\WEB-INF). EGL never overwrites a JSP.

**viewRootVar = "variableName"**

Identifies the name of a variable of type UIViewRoot, as is used to access the JSF component tree. For details, see *viewRootVar property*.

### Related concepts

“Overview of EGL properties” on page 64

“PageHandler” on page 223

“viewRootVar property” on page 230

**Related reference**

“PageHandler part in EGL source format” on page 785

## PageHandler field properties

The PageHandler field properties specify characteristics that are meaningful when a field is declared in a PageHandler part.

The properties are as follows:

- “action” on page 800
- “byPassValidation” on page 802
- “displayName” on page 807
- “displayUse” on page 808
- “help” on page 810
- “newWindow” on page 818
- “numElementsItem” on page 819
- “selectFromListItem” on page 823
- “selectType” on page 824
- “validationOrder” on page 831
- “value” on page 836

**Related concepts**

“Overview of EGL properties” on page 64

“PageHandler” on page 223

**Related tasks**

“Creating an EGL pageHandler part” on page 221

“Using the Quick Edit view for PageHandler code” on page 244

**Related reference**

“PageHandler part properties” on page 788

“PageHandler part in EGL source format” on page 785

“Page Designer support for EGL” on page 221

---

## pfKeyEquate

When you declare a form group that references a text form, the property *pfKeyEquate* specifies whether the keystroke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12.

If you accept the default value of *yes* for *pfKeyEquate*, your your logical expressions are able to reference only 12 of the function keys because (for example) PF2 is the same as PF14.

**Note:** Function keys on a PC keyboard are often *F* keys such as F1, but EGL uses the IBM *PF* terminology so that (for example) F1 is called PF1.

**Related concepts**

“FormGroup part” on page 183

**Related reference**

“FormGroup part in EGL source format” on page 603

---

## Primitive field-level properties

The next table lists the primitive field-level properties in EGL.

property	Description
@linkParameter (as described in @programLinkData)	Identifies the value to be placed in a particular input-record field of a receiving program when the user at a Web browser transfers control from one program of type VGWebTransaction to another.
@linkParms (as described in @programLinkData)	Identifies the values to be placed in an input record when the user at a Web browser transfers control from one program of type VGWebTransaction to another.
@programLinkData	Specifies details for transferring control from one program of type VGWebTransaction to another when the user at a Web browser submits a form or clicks a hypertext link.
@xsd	Provides a mapping of data types for use when EGL code interacts with a Web service: <ul style="list-style-type: none"><li>• In relation to an EGL Web service, establishes a set of validation rules that limit the kind of data that will be provided to the service at run time</li><li>• In relation to an interface that provides access to a Web service, ensures a valid mapping between an EGL data type and the type of data submitted to or received from the service</li></ul>
action	Identifies the code that is invoked when the user clicks the button or link.
alias	Is used in a program of type VGWebTransaction to relate a VGUI field and a JSP control ID. The property is useful in this case: <ul style="list-style-type: none"><li>• You migrated a program from VisualAge Generator; and</li><li>• The ID in the existing JSP is not valid in EGL.</li></ul>
align	Specifies the position of data in a variable field when the length of the data is smaller than the length of the field.
byPassValidation	Indicates whether EGL-based validation is bypassed when the user clicks the button or link.
color	Specifies the color of a field in a text form.
column	Refers to the name of the database table column that is associated with the field. The default is the name of the field.

property	Description
currency	Indicates whether to include a currency symbol before the value in a numeric field, with the exact position of the symbol determined by the <b>zeroFormat</b> property.
currencySymbol	Indicates which currency symbol to use when the property <b>currency</b> is in effect.
dateFormat	Identifies the format for dates.
	Specifies whether the field's modified data tag is set when the field is selected by a light pen or (for emulator sessions) by a cursor click.
displayName	Specifies the label that is displayed next to the control on a Web page.
displayNames	Specifies an array of labels, each of which is displayed next to the equivalent control in an array of controls. This property is used only in a program of type <b>VGWebTransaction</b> .
displayUse	Associates an EGL field with a user-interface control.
dliFieldName	Refers to the name of the DL/I database segment field that is associated with the field in the EGL record of type <b>DLISegment</b> . The default is the name of the EGL record field.
fieldLen	Specifies the number of single-byte characters that can be displayed in a text-form field.
fill	Indicates whether the user is required to enter data in each field position.
fillCharacter	Indicates what character fills unused positions in a text or print form or in <b>PageHandler</b> data.
help	Specifies the hover-help text that is displayed when the user places the cursor over the input field.
highlight	Specifies the special effect (if any) with which to display the field.
inputRequired	Indicates whether the user is required to place data in the field.
inputRequiredMsgKey	Identifies the message that is displayed if the field property <b>inputRequired</b> is set to <i>yes</i> and the user fails to place data into the field.
intensity	Specifies the strength of the displayed font.
isBoolean	Indicates that the field represents a Boolean value.
isDecimalDigit	Determines whether to check that the input value includes only decimal digits
isHexDigit	Determines whether to check that the input value includes only hexadecimal digits

property	Description
isNullable	Indicates whether the field can be set to null, as is appropriate if the table column associated with the field can be set to NULL.
isReadOnly	Indicates whether the field and related column should be omitted from the default SQL statements that write to the database or include a FOR UPDATE OF clause.
lineWrap	Indicates whether text can be wrapped onto a new line whenever wrapping is necessary to avoid truncating text.
lowerCase	Indicates whether to set alphabetic characters to lower case in the user's single-byte character input.
masked	Indicates whether a user-entered character will or will not be displayed.
maxLen	Specifies the maximum length of field text that is written to the database column.
minimumInput	Indicates the minimum number of characters that the user is required to place in the field, if the user places any data in the field.
minimumInputMsgKey	Identifies the message that is displayed if the user acts as follows: <ul style="list-style-type: none"> <li>Places data in the field; and</li> <li>Places fewer characters than the value specified in the property <b>minimumInputRequired</b>.</li> </ul>
modified	Indicates whether the program will consider the field to have been modified, regardless of whether the user changed the value.
needsSOSI	Indicates whether EGL does a special check when the user enters data of type MBCHAR on an ASCII device.
newWindow	Indicates whether to use a new browser window when the EGL runtime presents a Web page in response to the activity identified in the <b>action</b> property.
numElementsItem	Identifies a PageHandler field whose runtime value specifies the number of array elements to display.
numericFormat	Specifies the format of a numeric field in a Web page displayed by a program of type VGWebTransaction.
numericSeparator	Indicates whether to place a character in a number that has an integer component of more than 3 digits.
outline	Lets you draw lines at the edges of fields on any device that supports double-byte characters.

property	Description
pattern	Matches the user entered text against a specified pattern, for validation.
persistent	Indicates whether the field is included in the implicit SQL statements generated for the SQL record.
protect	Specifies whether the user can access the field.
runValidatorFromProgram	Is used in a program of type VGWebTransaction to indicate whether the validator function runs on the Web application server (in the UI record bean) or runs in the program that receives data from the user.
selectedIndexItem	Is used for a VGUI-field array and refers to the <i>selected index item</i> , which is a VGUI field whose value indicates two aspects of runtime processing: whether a control (such as a check box) is pre-selected when the Web page is displayed; and whether the user selected a control.
selectFromListItem	Identifies the array or DataTable column from which the user selects a value or values, which are then transferred to the array or primitive field being declared.
selectType	Indicates the kind of value that is retrieved into the array or primitive field being declared.
sign	Indicates the position in which a positive (+) or negative (-) sign is displayed when a number is placed in the field, whether from user input or from the program.
sqlDataCode	Identifies the SQL data type that is associated with the record field.
sqlVariableLen	Indicates whether trailing blanks and nulls in a character field are truncated before the EGL runtime writes the data to an SQL database.
timeFormat	Identifies the format for times.
timestampFormat	Identifies the format for timestamps that are displayed on a form or maintained in a PageHandler.
typeChkMsgKey	Identifies the message that is displayed if the input data is not appropriate for the field type.
uiType	Specifies the HTML tags to be created when a program of type VGWebTransaction issues a <b>show</b> or <b>converse</b> statement that displays a record of type VGUIRecord.
upperCase	Indicates whether to set alphabetic characters to upper case in the user's single-byte character input.

property	Description
validationOrder	Indicates when the field's validator function runs in relation to any other field's validator function.
validatorDataTable	Identifies a <i>validator table</i> , which is a <i>dataTable</i> part that acts as the basis of a comparison with user input.
validatorDataTableMsgKey	Identifies the message that is displayed if the user provides data that does not correspond to the requirements of the <i>validator table</i> , which is the table specified in the property <b>validatorDataTable</b> .
validatorFunction	Identifies a validator function, which is logic that runs after the EGL runtime does the elementary validation checks, if any.
validatorFunctionMsgKey	Identifies a message that is displayed if the validator function indicates an error.
validValues	Indicates a set of values that are valid for user input.
validValuesMsgKey	Identifies the message that is displayed if the field property <b>validValues</b> is set and the user places out-of-range data into the field.
value	Identifies a string literal that is displayed as the field content when a Web page is displayed.
zeroFormat	Specifies how zero values are displayed in numeric fields but not in fields of type MONEY.

## @programLinkData

The complex property **@programLinkData** is used in a program of type **VGWebTransaction** to specify details for transferring control to another **VGWebTransaction** program when the user submits a form or clicks a hypertext link.

The **@programLinkData** property fields and their types are as follows:

### programName String

Identifies the program to invoke when the user submits the form or clicks the hypertext link. An alias is used at generation time if you specify the alias or if you specify the name of a program part when that part has an alias and is in the current workspace.

If the **VGWebTransaction** part field is defined and accessible from the current workspace, the package name for the **VGWebTransaction** part is generated into the bean to be used at run time; otherwise, the package name for the program is derived from the **javaProperty** value of the application entry in the **hptLinkage** properties file.

### uiRecordName String

Identifies a **VGUI** record to send (if any). That record includes data from the Web form, from linkage parameters, or both. An alias is used at generation



time if you specify the alias or if you specify the name of a VGUIRecord part when that part has an alias and is in the current workspace.

If the VGUIRecord part specified for the uiRecordName field is defined and accessible from the current workspace during generation, the package name for the VGUIRecord part is generated into the bean to be used at run time; otherwise, the package name used for the program is used for the UI record.

#### **newWindow Boolean**

Indicates whether the Web page presented by the invoked program will be in a browser window different from the one from which the user submitted the form or clicked the hypertext link. Valid values are as follows:

##### **no (the default)**

The Web page will be in the same window. If the current form was presented by a **show** statement with a returning clause or by a **converse** statement, a new window disrupts the existing conversation with the user.

##### **yes**

The Web page will be in a different window

#### **@linkParms @linkParameter[]**

Identifies the values to be placed in the input record of the receiving program.

The **@linkParameter** property fields and their types are as follows:

##### **name STRING**

The name of a field in the input record of the program being invoked

##### **value literal**

A literal value to be passed

##### **valueRef STRING**

A field in the record being sent to the invoked program; the content of the record field is passed

You may not specify both value and valueRef in the same **@linkParameter** property.

#### **Related concepts**

"Overview of EGL properties" on page 64

## **@xsd**

The complex property **@xsd** provides a mapping of data types for use when EGL code interacts with a Web service:

#### **On the service side of the transmission**

Establishes a set of validation rules that limit the kind of data that will be provided to the service at run time

#### **On the client side**

Ensures a valid mapping between an EGL data type and the type of data submitted to or received from the service

The **@xsd** property affects an XSD element in the Web Service Description Language (WSDL) definition that is present on the service side of transmission (when an EGL Web Service is running) or that is used for runtime processing on the client side of transmission (when an EGL client is interacting with a Web service). The property is available for DataItem parts, for primitive-field

declarations within a Record part, and for Record parts; but in most cases you do not need to specify these properties because settings are provided by default or by an EGL wizard.

**Note:** When you are specifying the property for a Record part or when you are creating an interface to a Web service, the only property fields that have an effect are **elementName** and **namespace**.

The **@xsd** property fields reflect some of the XML-schema details that are described by the World Wide Web consortium (WC3):

<http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#facets>

The **@xsd** property fields and their types are as follows:

**elementName String**

Identifies the name of the xsd element in the WSDL definition. The default is either the name of the EGL part name (for DataItem parts and Record parts) or the name of an EGL variable (for primitive declarations in an EGL Record part).

**namespace String**

Specifies the namespace that represents the scope in which the element name resides. The default is the inversion of the EGL package name; if the package is named *a1.b2.c3*, for example, the default namespace is *c3.b2.a1*.

**base STRING**

Indicates the xsd type in the WSDL definition of the element.

**minLength STRING**

Specifies the minimum valid length of input. This property field is meaningful only in the following case:

- The xsd type is xsd:string or a value derived from xsd:string; and
- You are creating an EGL Web service.

In other cases, the property field is ignored.

**maxLength STRING**

Specifies the maximum valid length of input. This property field is meaningful only in the following case:

- The XSD type is xsd:string or a value derived from xsd:string; and
- You are creating an EGL Web service.

In other cases, the property field is ignored.

**pattern STRING**

Specifies a regular expression used to validate the input. This property field is meaningful only in the following case:

- The XSD type is xsd:string or a value derived from xsd:string; and
- You are creating an EGL Web service.

In other cases, the property field is ignored.

**enumeration STRING[ ]**

Specifies an array of valid values. This property field is meaningful only in the following case:

- The XSD type is xsd:string or a value derived from xsd:string; and
- You are creating an EGL Web service.

In other cases, the property field is ignored.

**whitespace STRING**

Indicates how whitespace is handled, as described in the W3C Web site mentioned earlier. This property field is meaningful only if you are creating an EGL Web service and is otherwise ignored.

**maxInclusive STRING**

Indicates the maximum value of a `xsd:string`, `xsd:time`, or `xsd:datetime` field when the value you specify is itself valid. This property field is meaningful only when you create an EGL Web service and is otherwise ignored.

**minInclusive STRING**

Indicates the minimum value of a `xsd:string`, `xsd:time`, or `xsd:datetime` field when the value you specify is itself valid. This property field is meaningful only when you create an EGL Web service and is otherwise ignored.

**maxExclusive STRING**

Indicates the maximum value of a `xsd:string`, `xsd:time`, or `xsd:datetime` field when the value you specify is not valid. This property field is meaningful only when you create an EGL Web service and is otherwise ignored.

**minExclusive STRING**

Indicates the minimum value of a `xsd:string`, `xsd:time`, or `xsd:datetime` field when the value you specify is not valid. This property field is meaningful only when you create an EGL Web service and is otherwise ignored.

For details on default mappings, see *Data conversions between WSDL and EGL*.

**Related concepts**

“Overview of EGL properties” on page 64

**Related reference**

“Data conversions between WSDL and EGL” on page 562

## action

When the EGL property **displayUse** is *button* or *hyperlink*, the property **action** identifies the code that is invoked when the user clicks the button or link in a Web page displayed by a PageHandler. The property has not effect for fields in VGUIRecords.

The value you assign to **action** is used as a default when you place the field (or a record that includes the field) on the Web Page in Page Designer.

The value of **action** is one of these kinds of string literals:

- The name of an event-handling function in the PageHandler
- A label that maps to a Web resource (for example, to a JSP) and that corresponds to a from-outcome attribute of a navigation-rule entry in the JSF Application Configuration Resource file
- The name of a method in a Java bean, in which case these rules apply:
  - The format is the bean name followed by a period and a method name
  - The bean name must relate to one of the managed bean-name entries in the JSF Application Configuration Resource file

If you do not specify a value for **action**, the user’s click of the field has the following effect:

- If the value of the property **displayUse** is *button*, validation occurs, after which JSF re-displays the same Web page.
- If the value of the property **displayUse** is *hyperlink*, no validation occurs, but JSF re-displays the same Web page.

#### Related concepts

“Overview of EGL properties” on page 64

“PageHandler” on page 223

#### Related tasks

“Binding a JavaServer Faces command component to an EGL PageHandler” on page 244

“Creating an EGL pageHandler part” on page 221

“Using the Quick Edit view for PageHandler code” on page 244

#### Related reference

“PageHandler field properties” on page 792

“PageHandler part properties” on page 788

“PageHandler part in EGL source format” on page 785

“Page Designer support for EGL” on page 221

## alias

The property **alias** is used in a program of type VGWebTransaction to relate a VGUI field with a JSP control ID. The property is useful in this case:

- You migrated a program from VisualAge Generator, and
- The ID in the existing JSP file is not valid in EGL.

You have no reason to set the **alias** property if the field name in the VGUI record is equivalent to the control ID, as is usually the case.

#### Related concepts

“Overview of EGL properties” on page 64

## align

The **align** property specifies the position of data in a variable field when the length of the data is smaller than the length of the field.

Values are of the enumeration **alignKind**:

#### left (the default for character data)

Place the data at the left of the field. Initial spaces are stripped and placed at the end of the field.

#### right (the default for numeric data)

Place the data at the right of the field. Trailing spaces are stripped and placed at the beginning of the field. This setting is required for numeric data that has a decimal position or sign.

#### center

Place the data at the center of the field. Leading and trailing spaces are stripped so that spacing is calculated based only on the other field content.

#### none

Do not justify the data. This setting is valid only for character data.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Console forms
- Print forms
- Text forms
- Web pages

On output, character and numeric data are affected by this property. On input, character data is affected by this property, but numeric data is always right-justified.

#### Related concepts

“Enumerations in EGL” on page 578

“Overview of EGL properties” on page 64

#### Related reference

“Formatting properties” on page 67

## byPassValidation

When the EGL property **displayUse** is *button* or *hyperlink*, the property **byPassValidation** indicates whether EGL-based validation is bypassed when the user clicks the button or link. You might want to bypass validation for better performance; for example, whenever the user clicks an Exit button.

The value you assign to **byPassValidation** is used as a default when you place the field (or a record that includes the field) on the Web Page in Page Designer.

The property affects only EGL-based validations, not those specified by JSF tags; for details, see *PageHandler*.

Values are of the enumeration **Boolean**:

#### no (the default)

The input fields are validated as usual

#### yes

The EGL runtime does not return user data to the PageHandler

#### Related concepts

“Enumerations in EGL” on page 578

“Overview of EGL properties” on page 64

“PageHandler” on page 223

#### Related tasks

“Binding a JavaServer Faces command component to an EGL PageHandler” on page 244

“Creating an EGL pageHandler part” on page 221

“Using the Quick Edit view for PageHandler code” on page 244

#### Related reference

“PageHandler field properties” on page 792

“PageHandler part properties” on page 788

“PageHandler part in EGL source format” on page 785

“Page Designer support for EGL” on page 221

## color

The **color** property specifies the color of a field in a text form. You can select any of these:

- black
- blue
- cyan
- defaultColor (the default)
- green
- magenta
- red
- white
- yellow

If you assign the value *defaultColor*, other conditions determine the displayed color, as shown in the next table.

Are all fields on the form assigned the value <i>defaultColor</i> ?	Value of <i>protect</i>	Value of <i>intensity</i>	Displayed color for a field assigned the value <i>defaultColor</i>
yes	<i>yes or skip</i>	not <i>bold</i>	blue
yes	<i>yes or skip</i>	<i>bold</i>	white
yes	<i>no</i>	not <i>bold</i>	green
yes	<i>no</i>	<i>bold</i>	red
no	any value	not <i>bold</i>	green
no	any value	<i>bold</i>	white

#### Related concepts

"Enumerations in EGL" on page 578

"Overview of EGL properties" on page 64

#### Related reference

"Field-presentation properties" on page 67

## column

The property **column** refers to the name of the database table column that is associated with the field. The default is the name of the field. The column and related field affect the default SQL statements, as described in *SQL support*.

For "*columnName*", substitute a quoted string; a variable of a character type; or a concatenation, as in this example:

```
column = "Column" + "01"
```

A special syntax applies if a column name is one of the following SQL reserved words:

- CALL
- COLUMNS
- FROM
- GROUP
- HAVING
- INSERT
- ORDER
- SELECT

- SET
- UPDATE
- VALUES
- WHERE

As shown in the following example, each of those names must be embedded in a doubled pair of quote marks, and each of the internal quote marks must be preceded with an escape character (\):

```
column = "\"SELECT\""
```

(A similar situation applies if you use of those reserved words as a table name.)

#### Related concepts

“Compatibility with VisualAge Generator” on page 532

“Record types and properties” on page 138

“SQL support” on page 277

“Fixed structure” on page 27

“Typedef” on page 28

#### Related tasks

“Retrieving SQL table data” on page 299

#### Related reference

“Field-presentation properties” on page 67

“add” on page 661

“close” on page 669

“Data initialization” on page 564

“delete” on page 673

“execute” on page 677

“get” on page 687

“get next” on page 701

“open” on page 722

“prepare” on page 736

“Primitive types” on page 34

“Record and file type cross-reference” on page 860

“replace” on page 738

“set” on page 742

“SQL data codes and EGL host variables” on page 874

“terminalID” on page 1075

“VAGCompatibility” on page 497

## currency

The **currency** property indicates whether to include a currency symbol before the value in a numeric field, with the exact position of the symbol determined by the **zeroFormat** property. The formatting of fields of type MONEY depends on the value of **strLib.defaultMoneyFormat** and is not affected by the **currency** property.

Values of the **currency** property are as follows:

#### No (the default)

Do not use a currency symbol.

#### Yes

Use the symbol specified in **currencySymbol**. If no value is specified there, use the default currency symbol.

The default currency symbol is determined by the machine locale.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Print forms
- Text forms
- Web pages

The property is used at input and output.

#### **Related concepts**

"Enumerations in EGL" on page 578

"Overview of EGL properties" on page 64

#### **Related reference**

"Formatting properties" on page 67

## **currencySymbol**

The **currencySymbol** property indicates which currency symbol to use when the property **currency** is in effect. The value is a string literal.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Print forms
- Text forms
- Web pages

The property is used at input and output.

#### **Related concepts**

"Enumerations in EGL" on page 578

"Overview of EGL properties" on page 64

#### **Related reference**

"Formatting properties" on page 67

## **dateFormat**

The **dateFormat** property identifies the format for dates.

Valid values are as follows:

*"pattern"*

The value of *pattern* consists of a set of characters, as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete date specification, but not from the middle.

#### **defaultDateFormat**

If specified for a page field, the value of **defaultDateFormat** is the date format given in the runtime Java locale. If specified for a form field, the default pattern is equivalent to selecting **systemGregorianCalendarFormat**.

#### **eurDateFormat**

The pattern "dd.MM.yyyy", which is the IBM European standard date format.



**isoDateFormat**

The pattern "yyyy-MM-dd", which is the date format specified by the International Standards Organization (ISO).

**jisDateFormat**

The pattern "yyyy-MM-dd", which is the Japanese Industrial Standard date format.

**usaDateFormat**

The pattern "MM/dd/yyyy", which is the IBM USA standard date format.

**systemGregorianCalendarFormat**

An 8- or 10-character pattern that includes dd (for numeric day), MM (for numeric month), and yy or yyyy (for numeric year), with characters other than d, M, y, or digits used as separators.

The format is in this Java runtime property:

```
vgj.datemask.gregorian.long.NLS
```

**NLS**

The NLS (national language support) code that is specified in the Java runtime property **vgj.nls.code**. The code is one of those listed in targetNLS. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java runtime properties (details)*.

**systemJulianDateFormat**

A 6- or 8-character pattern that includes DDD (for numeric day) and yy or yyyy (for numeric year), with characters other than D, y, or digits used as separators.

The format is in this Java runtime property:

```
vgj.datemask.julian.long.NLS
```

**NLS**

The NLS (national language support) code that is specified in the Java runtime property **vgj.nls.code**. The code is one of those listed in targetNLS. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java runtime properties (details)*.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Console forms
- Print forms
- Text forms
- Web pages

This property is used for both input and output, but not in the following cases:

- The field has decimal places, a currency symbol, a numeric separator, or a sign; or
- The field is of type DBCHAR, MBCHAR, or HEX; or
- The field is not long enough to contain a value that reflects the mask. For other details, see "Length considerations for dates" on page 807.

**Internal date formats**

When the user enters valid data, the date is converted from the format specified for the field to an internal format that is used for subsequent validation.

The internal format for a character date is the same as the system default format and includes separator characters.

For a numeric date, the internal formats are as follows:

- For a Gregorian short date, 00yyMMdd
- For a Gregorian long date, 00yyyyMMdd
- For a Julian short date, 0yyDDD
- For a Julian long date, 0yyyyDDD

### Length considerations for dates

In a form, the field length on the form must be greater than or equal to the length of the field mask that you specify. The length of the field must be long enough to hold the internal format of the date.

In a page field, the rules are as follows:

- The field length must be sufficient for the date mask you specify but can be longer
- In the case of a numeric field, the separator characters are excluded from the length calculation.

Examples are in the next table.

Format type	Example	Length of form field	Minimum length of page field (character type)	Valid length of page field (numeric type)
Short Gregorian	yy/MM/dd	8	8	6
Long Gregorian	yyyy/MM/dd	10	10	8
Short Julian	DDD-yy	6	6	5
Long Julian	DDD-yyyy	8	8	7

### I/O considerations for dates

Data entered into a variable field is checked to ensure that the date was entered in the format specified. The user does not need to enter the leading zeros for days and months, but can specify (for example) 8/5/1996 instead of 08/05/1996. The user who omits the separator characters, however, must enter all leading zeros.

#### Related concepts

“Java runtime properties” on page 431

“Overview of EGL properties” on page 64

#### Related reference

“Date, time, and timestamp format specifiers” on page 46

“Formatting properties” on page 67

“Java runtime properties (details)” on page 642

## displayName

The **displayName** property specifies the label that is displayed next to the field:

- In relation to a field in a PageHandler, you assign a string literal that is used as a default label when you place the field (or a record that includes the field) on the Web Page in Page Designer.

- In relation to VGUIRecord parts in programs of type VGWebTransaction, you assign a string literal that is used as the label of the control in the generated JSP file.

If you set **displayName** for a structure-field array in a VGUIRecord part, the following statements apply:

- If the string literal includes the newline character ("`\n`"), each successive line of text is the label for the sequentially next element in the array. An example of such a literal is as follows:  
`"label one:\nlabel two:\nlabel three:"`  
 In this example, the first array element receives the label **label one:**, the second receives the label **label two:**, and the third receives **label three:**.
- If you specify a string that does not include the newline character, that string is used as the label for each array element.
- If you specify a string that has fewer lines than are needed to match each element in the array, the last label that you provided is repeated for each array element that was not assigned a label.
- If you specify a string that has more lines than are needed, the extra labels are ignored.

#### Related concepts

"Overview of EGL properties" on page 64

"PageHandler" on page 223

#### Related tasks

"Associating an EGL record with a Faces JSP" on page 243

"Creating an EGL pageHandler part" on page 221

"Creating an EGL field and associating it with a Faces JSP" on page 242

"Using the Quick Edit view for PageHandler code" on page 244

#### Related reference

"PageHandler field properties" on page 792

"PageHandler part properties" on page 788

"PageHandler part in EGL source format" on page 785

"Page Designer support for EGL" on page 221

## displayName

Specifies an array of labels, each of which is displayed next to the equivalent control in an array of controls.

The value of this property is an array of string literals.

#### Related concepts

"Overview of EGL properties" on page 64

## displayUse

The **displayUse** property associates an EGL field with a user-interface control. The value you assign is used as a default when you place the field (or a record that includes the field) on the Web Page in Page Designer.

Values are of the enumeration **displayUseKind**:

#### button

The control has a button command tag

**secret**

The data is not visible to the user. This value is appropriate for passwords.

**hyperlink**

If the **action** property is the name of an event-handling function, the control has a hyperlink command tag. If the **action** property is a label, the control has a link tag. When the user clicks the link in either case, no validation occurs and no input data is returned.

**input**

The control accepts user input. Initially, the control may display a value provided by the PageHandler.

**table**

Data is within a table tag.

**output**

PageHandler field output, if any, is visible in the control.

**Related concepts**

"Enumerations in EGL" on page 578

"Overview of EGL properties" on page 64

"PageHandler" on page 223

**Related tasks**

"Associating an EGL record with a Faces JSP" on page 243

"Binding a JavaServer Faces command component to an EGL PageHandler" on page 244

"Creating an EGL pageHandler part" on page 221

"Creating an EGL field and associating it with a Faces JSP" on page 242

"Using the Quick Edit view for PageHandler code" on page 244

**Related reference**

"PageHandler field properties" on page 792

"PageHandler part properties" on page 788

"PageHandler part in EGL source format" on page 785

"Page Designer support for EGL" on page 221

**dliFieldName**

The property **dliFieldName** refers to the name of the DL/I database segment field that is associated with the field in the EGL record of type DLIsegment. The default is the name of the EGL record field.

The value of **dliFieldName** is folded to uppercase during generation to conform with DL/I requirements.

**Related concepts**

"Overview of EGL properties" on page 64

**fieldLen**

The property **fieldLen** specifies the number of single-byte characters that can be displayed in a text-form field. This value does not include the preceding attribute byte.

The value of **fieldLen** for numeric fields must be great enough to display the largest number that can be held in the field, plus (if the number has decimal places) a decimal point. The value of **fieldLen** for a field of type CHAR, DBCHAR,

MBCHAR, or UNICODE must be large enough to account for the double-byte characters, as well as any shift-in/shift-out characters.

The default **fieldLen** is the number of bytes needed to display the largest number possible for the primitive type, including all formatting characters.

#### Related concepts

“Overview of EGL properties” on page 64

#### Related reference

“Form part in EGL source format” on page 606

## fill

The **fill** property indicates whether the user is required to enter data in each field position. Valid values are *no* (the default) and *yes*.

#### Related concepts

“Text forms” on page 188

#### Related reference

“Validation properties” on page 68

“validationFailed()” on page 918

“DataTable part in EGL source format” on page 568

“verifyChkDigitMod10()” on page 1043

“verifyChkDigitMod11()” on page 1044

## fillCharacter

The **fillCharacter** property indicates what character fills unused positions in a text or print form or in PageHandler data. In addition, the property changes the effect of *set field full*, as described in *set*. The effect of this property is only at output.

The default is a space for numbers and a 0 for hex items. The default for character types depends on the medium:

- In text or print forms, the default is an empty string
- For PageHandler data, the default is blank for data of type CHAR or MBCHAR

In PageHandlers, the value of **fillCharacter** must be a space (as is the default) for items of type DBCHAR or UNICODE.

## help

The **help** property specifies the hover-help text that is displayed when the user places the cursor over the input field. The value you assign is used as a default when you place the EGL field (or a record that includes the EGL field) on the Web Page in Page Designer.

The value of this property is a string literal.

#### Related concepts

“Overview of EGL properties” on page 64

“PageHandler” on page 223

#### Related tasks

“Associating an EGL record with a Faces JSP” on page 243

“Creating an EGL pageHandler part” on page 221

“Creating an EGL field and associating it with a Faces JSP” on page 242  
“Using the Quick Edit view for PageHandler code” on page 244

#### Related reference

“PageHandler field properties” on page 792  
“PageHandler part properties” on page 788  
“PageHandler part in EGL source format” on page 785  
“Page Designer support for EGL” on page 221

## highlight

The **highlight** property specifies the special effect (if any) with which to display the field. Valid values are as follows:

#### **noHighLight (the default)**

Indicates that no special effect is to occur; specifically, no blink, reverse, or underline.

#### **underline**

Places an underline at the bottom of the field.

#### Related concepts

“Enumerations in EGL” on page 578  
“Overview of EGL properties” on page 64

#### Related reference

“Field-presentation properties” on page 67

## inputRequired

The **inputRequired** property indicates whether the user is required to place data in the field. Valid values are *no* (the default) and *yes*.

If the user does not place data in the field when the property value is *yes*, the EGL runtime displays a message, as described in relation to the field property

**inputRequiredMsgKey**.

#### Related concepts

“Text forms” on page 188

#### Related reference

“Validation properties” on page 68  
“validationFailed()” on page 918  
“DataTable part in EGL source format” on page 568  
“verifyChkDigitMod10()” on page 1043  
“verifyChkDigitMod11()” on page 1044

## inputRequiredMsgKey

The property **inputRequiredMsgKey** identifies the message that is displayed if the field property **inputRequired** is set to *yes* and the user fails to place data into the field.

The *message table* (the data table that contains the message) is identified in the program property **msgTablePrefix**. For details on the data-table name, see *DataTable part in EGL source format*.

The value of **inputRequiredMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

#### Related concepts

"Text forms" on page 188

#### Related reference

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## intensity

The **intensity** property specifies the strength of the displayed font. Valid values are as follows:

#### **normalIntensity (the default)**

Sets the field to be visible, without boldface.

#### **bold**

Causes the text to appear in boldface.

#### **dim**

Causes the text to appear with a lessened intensity, as appropriate when an input field is disabled.

#### **invisible**

Removes any indication that the field is on the form.

#### Related concepts

"Enumerations in EGL" on page 578

"Overview of EGL properties" on page 64

#### Related reference

"Field-presentation properties" on page 67

## isBoolean

The **isBoolean** property (formerly the **boolean** property) indicates that the field represents a Boolean value. The property restricts the valid field values and is useful in text and print forms and in PageHandlers, for input or output.

On a Web page associated with an EGL PageHandler, a boolean item is represented by a check box. On a form, the situation is as follows:

- The value of a numeric field is 0 (for false) or 1 (for true).
- The value of a character field is represented by a word or subset of a word that is national-language dependent. In English, for example, a boolean field of three or more characters has the value *yes* (for true) or *no* (for false), and a one-character boolean field value has the truncated value *y* or *n*.

Thespecific character values for *yes* and *no* are determined by the locale.

## isDecimalDigit

The **isDecimalDigit** property determines whether to check that the input value includes only decimal digits, which are as follows:

0123456789

Valid values are *no* (the default) and *yes*.

This property applies only to character fields.

### Related concepts

"Text forms" on page 188

### Related reference

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## isHexDigit

The **isHexDigit** property determines whether to check that the input value includes only hexadecimal digits, which are as follows:

0123456789abcdefABCDEF

Valid values are *no* (the default) and *yes*.

This property applies only to character fields.

### Related concepts

"Text forms" on page 188

### Related reference

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## isNullable

The property **isNullable** indicates whether the field can be set to NULL. The property is available only for primitive fields in a non-fixed record or for structure fields in an SQL record. Valid values are *no* (the default) and *yes*.

If a given field in an SQL record is nullable, the following features are available:

- The field can accept a NULL value from an SQL database table.
- A **set** statement can null the field, as described in *set*. The effect of setting a field to null is also to initialize the field, as described in *Data initialization*.
- An **if** statement can test whether the field is set to null.

If you are generating code with the build descriptor option **itemsNullable** set to YES, the effect is as follows:

- All primitive variables are nullable, whether they are standalone variables or fields in non-fixed records



- All structure fields in SQL records are nullable

#### **Related concepts**

“Compatibility with VisualAge Generator” on page 532  
 “Record types and properties” on page 138  
 “SQL support” on page 277  
 “Fixed structure” on page 27  
 “Typedef” on page 28

#### **Related tasks**

“Retrieving SQL table data” on page 299

#### **Related reference**

“Field-presentation properties” on page 67  
 “add” on page 661  
 “close” on page 669  
 “Data initialization” on page 564  
 “delete” on page 673  
 “execute” on page 677  
 “get” on page 687  
 “get next” on page 701  
 “itemsNullable” on page 482  
 “open” on page 722  
 “prepare” on page 736  
 “Primitive types” on page 34  
 “Record and file type cross-reference” on page 860  
 “replace” on page 738  
 “set” on page 742  
 “SQL data codes and EGL host variables” on page 874  
 “terminalID” on page 1075  
 “VAGCompatibility” on page 497

## **isReadOnly**

The property **isReadOnly** indicates whether the field and related column should be omitted from the default SQL statements that write to the database or include a FOR UPDATE OF clause. The default value is *no*; but EGL treats the structure field as “read only” in these situations:

- The property **key** of the SQL record indicates that the column that is associated with the record field is a key column; or
- The SQL record part is associated with more than one table; or
- The SQL column name is an expression.

#### **Related concepts**

“Compatibility with VisualAge Generator” on page 532  
 “Record types and properties” on page 138  
 “SQL support” on page 277  
 “Fixed structure” on page 27  
 “Typedef” on page 28

#### **Related tasks**

“Retrieving SQL table data” on page 299

#### **Related reference**

“Field-presentation properties” on page 67  
 “add” on page 661

“close” on page 669  
“Data initialization” on page 564  
“delete” on page 673  
“execute” on page 677  
“get” on page 687  
“get next” on page 701  
“open” on page 722  
“prepare” on page 736  
“Primitive types” on page 34  
“Record and file type cross-reference” on page 860  
“replace” on page 738  
“set” on page 742  
“SQL data codes and EGL host variables” on page 874  
“terminalID” on page 1075  
“VAGCompatibility” on page 497

## lineWrap

The EGL property **lineWrap** indicates whether text can be wrapped onto a new line whenever wrapping is necessary to avoid truncating text.

Valid values are of the enumeration **lineWrapType**:

### **character (the default)**

The text in a field will not be split at a white space.

### **compress**

The text in a field of type ConsoleField will be split at a white space; but when the user leaves the field (either by navigating to another field or by pressing Esc), any extra spaces that are used to wrap text are removed. This value is valid only in console fields.

### **word**

If possible, the text in a field will be split at a white space.

The property **lineWrap** is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Console forms
- Print forms
- Text forms
- Web pages

The property affects input and output.

### **Related concepts**

“Enumerations in EGL” on page 578  
“Overview of EGL properties” on page 64

### **Related reference**

“Formatting properties” on page 67

## lowerCase

The **lowerCase** property indicates whether to set alphabetic characters to lower case in the user’s single-byte character input. Values are as follows:

### **no (the default)**

Do not set the user’s input to lower case.

**yes**

Set the user's input to lower case.

## **masked**

The **masked** property indicates whether a user-entered character will or will not be displayed. This property is used for entering passwords. Values are as follows:

**no (the default)**

The user-entered character will be displayed.

**yes**

The user-entered character will not be displayed.

## **maxLen**

The property **maxLen** specifies the maximum length of field text that is written to the database column. Whenever possible, the default value for this property is the length of the field; but if the field is of type **STRING**, no default value exists.

### **Related concepts**

"Compatibility with VisualAge Generator" on page 532

"Record types and properties" on page 138

"SQL support" on page 277

"Fixed structure" on page 27

"Typedef" on page 28

### **Related tasks**

"Retrieving SQL table data" on page 299

### **Related reference**

"Field-presentation properties" on page 67

"add" on page 661

"close" on page 669

"Data initialization" on page 564

"delete" on page 673

"execute" on page 677

"get" on page 687

"get next" on page 701

"open" on page 722

"prepare" on page 736

"Primitive types" on page 34

"Record and file type cross-reference" on page 860

"replace" on page 738

"set" on page 742

"SQL data codes and EGL host variables" on page 874

"terminalID" on page 1075

"VAGCompatibility" on page 497

## **minimumInput**

The **minimumInput** property indicates the minimum number of characters that the user is required to place in the field, if the user places any data in the field. The default is 0.

If the user places fewer than the minimum number of characters, EGL run time displays a message, as described in relation to the field property

**minimumInputMsgKey**.

**Related concepts**

"Text forms" on page 188

**Related reference**

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## minimumInputMsgKey

The property **minimumInputMsgKey** identifies the message that is displayed if the user acts as follows:

- Places data in the field; and
- Places fewer characters than the value specified in the property **minimumInputRequired**.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **minimumInputMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

**Related concepts**

"Text forms" on page 188

**Related reference**

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## modified

Indicates whether the program will consider the field to have been modified, regardless of whether the user changed the value. For details, see *Modified data tag and modified property*.

The default is *no*.

**Related concepts**

"Modified data tag and modified property" on page 191

"Overview of EGL properties" on page 64

**Related reference**

"Form part in EGL source format" on page 606

## needsSOSI

The **needsSOSI** property is used only for a *multibyte field* (a field of type MBCHAR) and indicates whether EGL does a special check when the user enters data of type MBCHAR on an ASCII device. Valid values are *yes* (the default) and *no*. The check determines whether the input can be converted properly to the host SO/SI format.

The property is useful because, during conversion, trailing blanks are deleted from the end of a multibyte string to allow for insertion of SO/SI delimiters around each substring of double-byte characters. For a proper conversion, the form field must have at least two blanks for each double-byte string in the multibyte value.

If **needsSOSI** is set to *no*, the user can fill the input field, in which case the conversion truncates data without warning.

If **needsSOSI** is set to *yes*, however, the result is as follows when the user enters multibyte data:

- The value is accepted as is because enough blanks are provided; or
- The value is truncated, and the user receives a warning message.

Set **needsSOSI** to *yes* if the user's ASCII input of multibyte data may be used on a z/OS or iSeries system.

### Related concepts

"Text forms" on page 188

### Related reference

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## newWindow

The property **newWindow** indicates whether to use a new browser window when the EGL runtime presents a Web page in response to the activity identified in the **action** property.

Values are of the enumeration **Boolean**:

### No (the default)

The current browser window is used to display the page

### Yes

A new browser window is used.

If the **action** property is not specified, the current browser window is used to display the page.

### Related concepts

"Enumerations in EGL" on page 578

"Overview of EGL properties" on page 64

"PageHandler" on page 223

### Related tasks

- “Associating an EGL record with a Faces JSP” on page 243
- “Binding a JavaServer Faces command component to an EGL PageHandler” on page 244
- “Creating an EGL pageHandler part” on page 221
- “Creating an EGL field and associating it with a Faces JSP” on page 242
- “Using the Quick Edit view for PageHandler code” on page 244

### Related reference

- “action” on page 800
- “PageHandler field properties” on page 792
- “PageHandler part properties” on page 788
- “PageHandler part in EGL source format” on page 785
- “Page Designer support for EGL” on page 221

## numElementsItem

When set on a structure-field array, the property **numElementsItem** identifies a PageHandler field whose runtime value specifies the number of array elements to display. The property is used only for output and is meaningful only if set on a fixed-record structure field that has an occurs value greater than 1.

The value of **numElementsItem** is a string literal that identifies the name of a PageHandler field. The property is not valid for a dynamic array, which includes an indicator of how many elements are in use; for details, see *Arrays*.

### Related concepts

- “Fixed record parts” on page 136
- “Overview of EGL properties” on page 64
- “PageHandler” on page 223

### Related tasks

- “Associating an EGL record with a Faces JSP” on page 243
- “Creating an EGL pageHandler part” on page 221
- “Creating an EGL field and associating it with a Faces JSP” on page 242
- “Using the Quick Edit view for PageHandler code” on page 244

### Related reference

- “Arrays” on page 75
- “PageHandler field properties” on page 792
- “PageHandler part properties” on page 788
- “PageHandler part in EGL source format” on page 785
- “Page Designer support for EGL” on page 221

## numericFormat

The property **numericFormat** specifies the format of a numeric field.

The value is a string that can be used for input or output, but is not enabled in VGUIRecord fields, which are used in programs of type VGWebTransaction.

Valid characters are as follows:

- # A placeholder for a digit.
- \* Use an asterisk (\*) as the fill character for a leading zero.
- & Use a zero as the fill character for a leading zero.
- # Use a space as the fill character for a leading zero.

- < Left justify the number.
- , Use a locale-dependent numeric separator unless the position contains a leading zero.
- . Use a locale-dependent decimal point.
- Use a minus sign (-) for values less than 0; use a space for values greater than or equal to 0.
- + Use a minus sign for values less than 0; use a plus sign (+) for values greater than or equal to 0.
- ( Precede negative values with a left parenthesis, as appropriate in accounting.
- ) Place a right parenthesis after a negative value, as appropriate in accounting.
- \$ Precede the value with the locale-dependent currency symbol.
- @ Place the locale-dependent currency symbol after the value.

#### Related concepts

“Overview of EGL properties” on page 64

## numericSeparator

The **numericSeparator** property indicates whether to place a character in a number that has an integer component of more than 3 digits. If the numeric separator is a comma, for example, one thousand is shown as *1,000* and one million is shown as *1,000,000*. Values are as follows:

#### no (the default)

Do not use a numeric separator.

#### yes

Use a numeric separator.

The default is determined by the machine locale.

## runValidatorFromProgram

The property **runValidatorFromProgram** is meaningful only if the field is in a VGUI record. The property indicates whether the function referenced in the **validatorFunction** property runs on the Web application server (in the UI record bean) or runs in the program that receives data from the user.

The value is of enumeration **Boolean**:

#### no (the default)

The function runs on the Web application server.

#### yes

The function runs in the program.

Validate fields in the program only if validation requires access to program variables or to other resources that are not available on the Web application server.

#### Related concepts

“Overview of EGL properties” on page 64

## outline

The **outline** property lets you draw lines at the edges of fields on any device that supports double-byte characters. Valid values are as follows:

**box**

Draw lines to create a box around the field content

**noOutline (the default)**

Draw no lines

In addition, you can specify any or all of the components of a box. In this case, place brackets around one or more values, with each value separated from the next by a comma, as in this example:

```
outline = [left, over, right, under]
```

The partial values are as follows:

**left**

Draw a vertical line at the left edge of the field

**over**

Draw a horizontal line at the top edge of the field

**right**

Draw a vertical line at the right edge of the field

**under**

Draw a horizontal line at the bottom edge of the field

The content of each form field is preceded by an attribute byte. Be aware that you cannot place an attribute byte in the last column of a form and expect an outline value to appear in the next column, which is beyond the form's edge. (The field does not wrap to the next line.) Similarly, you cannot place an attribute byte in the first column of a form and expect the outline value to appear in that column; the outline value can appear only in the next column.

**Related concepts**

"Enumerations in EGL" on page 578

"Overview of EGL properties" on page 64

**Related reference**

"Field-presentation properties" on page 67

## **pattern**

Matches the user entered text against a specified pattern, for validation.

**Related concepts**

"Overview of EGL properties" on page 64

**Related reference**

"Form part in EGL source format" on page 606

## **persistent**

The property **persistent** indicates whether the field is included in the implicit SQL statements generated for the SQL record. If the value is *yes*, an error occurs at run time in this case:

- Your code relies on an implicit SQL statement; and
- No column matches the value of the field-specific **column** property. (The default value is the field name.)



Set **persistent** to *no* if you want to associate a temporary program variable with an SQL row without having a corresponding column for the variable in the database. You might want a variable, for example, to indicate whether the program has modified the row.

For details on implicit SQL statements, see *SQL support*.

#### **Related concepts**

"Compatibility with VisualAge Generator" on page 532

"Record types and properties" on page 138

"SQL support" on page 277

"Fixed structure" on page 27

"Typedef" on page 28

#### **Related tasks**

"Retrieving SQL table data" on page 299

#### **Related reference**

"Field-presentation properties" on page 67

"add" on page 661

"close" on page 669

"Data initialization" on page 564

"delete" on page 673

"execute" on page 677

"get" on page 687

"get next" on page 701

"open" on page 722

"prepare" on page 736

"Primitive types" on page 34

"Record and file type cross-reference" on page 860

"replace" on page 738

"set" on page 742

"SQL data codes and EGL host variables" on page 874

"terminalID" on page 1075

"VAGCompatibility" on page 497

## **protect**

Specifies whether the user can access the field. Valid values are as follows:

#### **no (the default for variable fields)**

Sets the field so that the user can overwrite the value in it.

#### **skip (the default for constant fields)**

Sets the field so that the user cannot overwrite the value in it. In addition, the cursor skips the field in either of these cases:

- The user is working on the previous field in the tab order and either presses **Tab** or fills that previous field with content; or
- The user is working on the next field in the tab order and presses **Shift Tab**.

#### **yes**

Sets the field so that the user cannot overwrite the value in it.

#### **Related concepts**

"Overview of EGL properties" on page 64

### Related reference

“Form part in EGL source format” on page 606

## selectedIndexItem

The property **selectedIndexItem** is used for a VGUI-field array and refers to the *selected index item*, which is a VGUI field whose value indicates two aspects of runtime processing:

- Whether a control (such as a check box) is pre-selected when the Web page is displayed
- Whether the user selected a control

The field to which the property refers must be of type NUM and must be without decimal places, but may be an array.

Consider the following case:

- A VGUI record field is an array and has a **selectedIndexItem** property that refers to a field (the selected index item) that is not an array
- The uiType property is input or inputOutput

In this case, the generated Web page contains radio buttons from which the user can choose only one value. The selected index item identifies the radio button by number, with the number 1 referring to the first radio button.

If the selected index item is an array in the same case, the generated Web page contains a set of checkboxes, and the following is true:

- When data is being prepared for display, the array contains the indexes of the entries to be set as pre-selected
- When data is returned, the array contains the indexes of the entries that the user selected. For example, if the user selected displayed entries with indexes 1, 3, and 5, for example, the first element of the array contains 1, the second element contains 3, the third contains 5, and the rest contain 0.

The generated HTML is different if the uiType property is output:

- If the selected index item is not an array, the generated Web page contains a pair of structures—the item label, which is displayed in boldface, and an HTML `<SELECT>` structure (seen as a drop-down list), from which the user can choose one value. If the selected index item has a value of 2, for example, the second entry is pre-selected.
- If the selected index item is an array, the generated Web page contains a similar pair of structures, but the user can choose multiple values. If the value of the first element in the selected index item is 1 and the value of element 2 is 3, the browser displays a drop-down list, and the first and third entries are pre-selected.

### Related concepts

“Overview of EGL properties” on page 64

## selectFromListItem

The property **selectFromListItem** identifies the array or DataTable column from which the user selects a value or values, which are then transferred to the array or primitive field being declared. The value you assign to **selectFromListItem** is used as a default when you place the array or primitive field on the Web Page in Page Designer.

The value of property **selectFromListItem** is a string literal that identifies the source array or DataTable column.

If you specify this property when declaring an array, the user is allowed to select multiple values. If you specify this property when declaring a primitive field, the user can select only one value.

Any value received from the user must correspond to one of these types:

- The content of the array element or DataTable column that the user selected; or
- An array or DataTable index, which is an integer that identifies which element or column was selected. The index ranges from 1 to the number of elements available.

The property **selectType** indicates the type of value to receive, whether the content selected by the user or an index into an array or column.

#### **Related concepts**

“DataTable” on page 176

“Overview of EGL properties” on page 64

“PageHandler” on page 223

#### **Related tasks**

“Associating an EGL record with a Faces JSP” on page 243

“Creating an EGL pageHandler part” on page 221

“Creating an EGL field and associating it with a Faces JSP” on page 242

“Using the Quick Edit view for PageHandler code” on page 244

#### **Related reference**

“Arrays” on page 75

“PageHandler field properties” on page 792

“PageHandler part properties” on page 788

“PageHandler part in EGL source format” on page 785

“Page Designer support for EGL” on page 221

“selectType”

## **selectType**

The property **selectType** indicates the kind of value that is retrieved into the array or primitive field being declared. The value you assign is used as a default when you place the array or primitive field on the Web Page in Page Designer.

The value is of enumeration **selectTypeKind**:

#### **index (the default)**

The array or primitive field being declared will receive indexes in response to a user selection. In this case, the array or primitive field must be of a numeric type.

#### **value**

The array or primitive field being declared will receive the user’s selection value. In this case, the item can be of any type.

For background information, see the property **selectFromListItem**.

### Related concepts

"DataTable" on page 176

"Overview of EGL properties" on page 64

"PageHandler" on page 223

### Related tasks

"Associating an EGL record with a Faces JSP" on page 243

"Creating an EGL pageHandler part" on page 221

"Creating an EGL field and associating it with a Faces JSP" on page 242

"Using the Quick Edit view for PageHandler code" on page 244

### Related reference

"Arrays" on page 75

"PageHandler field properties" on page 792

"PageHandler part properties" on page 788

"PageHandler part in EGL source format" on page 785

"Page Designer support for EGL" on page 221

"selectFromListItem" on page 823

## sign

The **sign** property indicates how to format a field to indicate that a numeric value is positive or negative. The property is meaningful whether the value is from user input or from EGL-generated code, and valid values are as follows:

### leading (the default value)

A sign is displayed to the left of the first digit in the number, with the exact position of the sign determined by the **zeroFormat** property (unless the primitive type for the field is MONEY)

### trailing

A sign is displayed immediately to the right of the last digit in the number

### none

A sign is not displayed

### parens

A negative number is displayed as a positive number within parentheses, as appropriate when the field contains a monetary value in an accounting application

### Related reference

"Primitive field-level properties" on page 793

"zeroFormat" on page 837

## sqlDataCode

The value of property **sqlDataCode** is a number that identifies the SQL data type that is associated with the record field. The data code is used by the database management system when you access that system at declaration time, validation time, or generated-program run time.

The property **sqlDataCode** is available only if you have set up your environment for VisualAge Generator compatibility. For details, see *Compatibility with VisualAge Generator*.

The default value depends on the primitive type and length of the record field, as shown in the next table. For other details, see *SQL data codes*.

EGL primitive type	Length	SQL data code
BIN	4	501
	9	497
CHAR	<=254	453
	>254 and <=4000	449
	>4000	457
DBCHAR	<=127	469
	>127 and <=2000	465
	>2000	473
DECIMAL	any	485
HEX	any	481
UNICODE	<=127	469
	>127 and <=2000	465
	>2000	473

#### Related concepts

“Compatibility with VisualAge Generator” on page 532

“Record types and properties” on page 138

“SQL support” on page 277

“Fixed structure” on page 27

“Typedef” on page 28

#### Related tasks

“Retrieving SQL table data” on page 299

#### Related reference

“Field-presentation properties” on page 67

“add” on page 661

“close” on page 669

“Data initialization” on page 564

“delete” on page 673

“execute” on page 677

“get” on page 687

“get next” on page 701

“open” on page 722

“prepare” on page 736

“Primitive types” on page 34

“Record and file type cross-reference” on page 860

“replace” on page 738

“set” on page 742

“SQL data codes and EGL host variables” on page 874

“terminalID” on page 1075

“VAGCompatibility” on page 497

## sqlVariableLen

The value of property **sqlVariableLen** (formerly the **sqlVar** property) indicates whether trailing blanks and nulls in a character field are truncated before the EGL runtime writes the data to an SQL database. This property has no effect on non-character data.

Specify *yes* if the corresponding SQL table column is a varchar or vargraphic SQL data type.

#### **Related concepts**

"Compatibility with VisualAge Generator" on page 532  
"Record types and properties" on page 138  
"SQL support" on page 277  
"Fixed structure" on page 27  
"Typedef" on page 28

#### **Related tasks**

"Retrieving SQL table data" on page 299

#### **Related reference**

"Field-presentation properties" on page 67  
"add" on page 661  
"close" on page 669  
"Data initialization" on page 564  
"delete" on page 673  
"execute" on page 677  
"get" on page 687  
"get next" on page 701  
"open" on page 722  
"prepare" on page 736  
"Primitive types" on page 34  
"Record and file type cross-reference" on page 860  
"replace" on page 738  
"set" on page 742  
"SQL data codes and EGL host variables" on page 874  
"terminalID" on page 1075  
"VAGCompatibility" on page 497

## **timeFormat**

The **timeFormat** property identifies the format for times.

Valid values are as follows:

*"pattern"*

The value of *pattern* consists of a set of characters as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete time specification, but not from the middle.

#### **defaultTimeFormat**

The default value in a Java environment is set by the Java locale.

#### **eurTimeFormat**

The pattern *HH.mm.ss*, which is the IBM European standard time format.

#### **isoTimeFormat**

The pattern *HH.mm.ss*, which is the time format specified by the International Standards Organization (ISO).

#### **jisTimeFormat**

The pattern *HH:mm:ss*, which is the Japanese Industrial Standard time format.

#### **usaTimeFormat**

The pattern *hh:mm AM*, which is the IBM USA standard time format.

The property is available in DataItem parts and is meaningful for fields that appear in the following contexts:

- Console forms
- Print forms
- Text forms
- Web pages

The property is used for both input and output, but not in the following cases:

- The field has decimal places, a currency symbol, a numeric separator, or a sign; or
- The field is of type DBCHAR, MBCHAR, or HEX; or
- The field is not long enough to contain a value that reflects the mask. For other details, see *Length considerations for times*.

### Length considerations for times

In a form, the field length must match the length of the time mask you specify. In a page field, the rules are as follows:

- The item length must be sufficient for the time mask you specify but can be longer
- In the case of a numeric item, the separator characters are excluded from the length calculation.

### I/O considerations for times

Data entered into a variable field is checked to ensure that the time was entered in the format specified. The user does not need to enter the leading zeros for hours, minutes, and second, but can specify (for example) 8:15 instead of 08:15. The user who omits the separator characters, however, must enter all leading zeros.

A time stored in internal format is not recognized as a time, but simply as data. If a 6-character time field is moved to a character item of length 10, for example, EGL pads the destination field with blanks. When the 6-character value is presented on a form, however, the time is converted from its internal format, as appropriate.

### Related concepts

"Java runtime properties" on page 431

"Overview of EGL properties" on page 64

### Related reference

"Date, time, and timestamp format specifiers" on page 46

"Formatting properties" on page 67

"Java runtime properties (details)" on page 642

## timestampFormat

The **timestampFormat** property identifies the format for timestamps that are displayed on a form or maintained in a PageHandler.

Valid values are as follows:

*"pattern"*

The value of *pattern* consists of a set of characters as described in *Date, time, and timestamp format specifiers*.

Characters may be dropped from the beginning or end of a complete timestamp specification, but not from the middle.

**defaultTimeStampFormat**

In a Java environment, the default is set by the Java locale.

**db2TimestampFormat**

The pattern *yyyy-MM-dd-HH.mm.ss.ffffff*, which is the IBM DB2 default timestamp format.

**odbcTimestampFormat**

The pattern *yyyy-MM-dd HH:mm:ss.ffffff*, which is the ODBC timestamp format.

**Related concepts**

"Java runtime properties" on page 431

"Overview of EGL properties" on page 64

**Related reference**

"Date, time, and timestamp format specifiers" on page 46

"Java runtime properties (details)" on page 642

## typeChkMsgKey

The property **typeChkMsgKey** identifies the message that is displayed if the input data is not appropriate for the field type.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **typeChkMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

**Related concepts**

"Text forms" on page 188

**Related reference**

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## uiType

The property **uiType** specifies the HTML tags to be created when a program of type **VGWebTransaction** issues a **show** or **converse** statement that displays a record of type **VGUIRecord**.

Valid values are of the enumeration **UITypeKind**:

**hidden**

Causes the record field to be translated to one or more HTML **<INPUT>** tags of type **HIDDEN**. The value that you assign programmatically to a record field in this case is available to any program invoked from the Web page, but is visible to the user only if the user reviews the Web-page source by way of



browser-specific keystrokes. If you want to prevent the user from seeing a value, do not use the `uiType` value *hidden*, but consider using the `uiType` value *none*.

#### **input**

Causes the record field to be translated to an HTML tag or tags that allow the user to assign values. The type of HTML tags according to other declaration-time settings.

EGL runtime performs input edits on values for which the `uiType` is `input`. Although the translated tags can display values assigned programmatically, EGL runtime does not perform output formatting on those values.

#### **inputOutput**

Causes the record field to be translated to an HTML tag or tags that have initial display values, which the user can change. The type of HTML tags according to other declaration-time settings.

EGL runtime performs both output formatting and input edits on values for which the `uiType` is `inputOutput`.

#### **none**

Causes the record field to be excluded from the HTML sent to the browser. In most cases, the field is available on tiers 2 and 3.

You can use the record field in an edit function that runs on tier 2, but only if the VGUI record is presented either by a **converse** statement or by a **show** statement with a *returning to* clause.

If a VGUI record is presented by a **show** statement without a *returning to* clause, you cannot store the record field on tier 2 for use after the user submits the page, because after the user submits the page, the UI record bean and UI record object are created with data from the browser and only from the browser.

If a record field for which the `uiType` is `none` is used as a link parameter in a field for which the `uiType` is `submit`, `submitBypass`, or `programLink`, EGL runtime treats the `uiType` `none` as `uiType` `hidden` and includes the generated field in the HTML.

#### **output**

Causes the value of the record field to be placed in the HTML. Some aspects of the HTML output vary according to other declaration-time settings.

The user cannot type data to change an output value on the Web page but in some cases can select one or more output values from a list, in which case the selections are made available to the program.

EGL runtime performs output formatting on fields for which the `uiType` is `output`.

#### **programLink**

Causes the record field to be translated to an HTML `<A>` tag, which is displayed as a hypertext link. You use the property **@programLinkData** to specify a `VGWebTransaction` program that will be invoked if the user clicks the hypertext link.

#### **submit**

Causes the record field to be translated to an HTML `<INPUT>` tag of type `SUBMIT`. If the user clicks the `SUBMIT` button that results from that tag, EGL runtime on tier 2 performs edits on the user's input and (if the edits succeed)

sends the user data (including the SUBMIT button value) to the program on tier 3. The button value is stored in the submit value field.

#### **submitBypass**

Causes the record field to be translated to an HTML <INPUT> tag of type SUBMIT. If the user clicks the SUBMIT button that results from that tag, the button value is stored in the submit value field and is made available to the program on tier 3. The rest of the user's input is ignored.

The primary use of a field for which uiType is submitBypass is to define an Exit button.

#### **uiForm**

Causes the record field to be translated to an HTML <FORM> structure, which is separate from the default HTML <FORM> structure that is provided when the Web transaction presents a Web page.

You use the property **@programLinkData** to specify a VGWebTransaction program that will be invoked if the user clicks a SUBMIT button from within the derived <FORM> structure. The data submitted to that program can include data from the user as well as data received from the program that presented the Web page.

#### **Related concepts**

"Overview of EGL properties" on page 64

## **upperCase**

The **upperCase** property indicates whether to set alphabetic characters to upper case in the user's single-byte character input.

This property is useful in forms and in PageHandlers.

The values of **upperCase** are as follows:

#### **No (the default)**

Do not set the user's input to upper case.

#### **Yes**

Set the user's input to upper case.

## **validationOrder**

The property **validationOrder** indicates when the field's validator function runs in relation to any other field's validator function. The property is important if the validation of one field depends on the previous validation of another.

The value is a literal integer.

Validation occurs first for any fields for which you specified a value for the property **validationOrder**, and the items with the lowest-numbered values are validated first. Validation then occurs for any items for which you did not specify a value for **validationOrder**, and the order of validation in this case is the order in which the fields are defined in the PageHandler.

#### **Related concepts**

"Overview of EGL properties" on page 64

"PageHandler" on page 223

### Related tasks

“Creating an EGL pageHandler part” on page 221

“Creating an EGL field and associating it with a Faces JSP” on page 242

“Using the Quick Edit view for PageHandler code” on page 244

### Related reference

“PageHandler field properties” on page 792

“PageHandler part properties” on page 788

“PageHandler part in EGL source format” on page 785

“Page Designer support for EGL” on page 221

## validatorDataTable

The **validatorDataTable** property (formerly the **validatorTable** property) identifies a *validator table*, which is a *dataTable* part that acts as the basis of a comparison with user input. Use of a validator table occurs after the EGL runtime does the elementary validation checks, if any. Those elementary checks are described in relation to the following properties:

- **inputRequired**
- **isDecimalDigit**
- **isHexDigit**
- **minimumInput**
- **needsSOSI**
- **validValues**

All checks precede use of the **validatorFunction** property, which specifies a validation function that does cross-value validation.

You can specify a validator table that is of any of the following types, as described in *DataTable part in EGL source format*:

### matchInvalidTable

Indicates that the user’s input must be different from any value in the first column of the data table.

### matchValidTable

Indicates that the user’s input must match a value in the first column of the data table.

### rangeChkTable

Indicates that the user’s input must match a value that is between the values in the first and second column of at least one data-table row. (The range is inclusive; the user’s input is also valid if it matches a value in the first or second column of any row.)

If validation fails, the displayed message is based on the value of the property **validatorDataTableMsgKey**.

### Related concepts

“Text forms” on page 188

### Related reference

“Validation properties” on page 68

“validationFailed()” on page 918

“DataTable part in EGL source format” on page 568  
“verifyChkDigitMod10()” on page 1043  
“verifyChkDigitMod11()” on page 1044

## validatorDataTableMsgKey

The property **validatorDataTableMsgKey** (formerly the **validatorTableMsgKey** property) identifies the message that is displayed if the user provides data that does not correspond to the requirements of the *validator table*, which is the table specified in the property **validatorDataTable**.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the message-table name, see *DataTable part in EGL source format*.

The value of **validatorDataTableMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

### Related concepts

“Text forms” on page 188

### Related reference

“Validation properties” on page 68  
“validationFailed()” on page 918  
“DataTable part in EGL source format” on page 568  
“verifyChkDigitMod10()” on page 1043  
“verifyChkDigitMod11()” on page 1044

## validatorFunction

The **validatorFunction** property (formerly the **validator** property) identifies a validator function, which is logic that runs after the EGL runtime does the elementary validation checks, if any. Those checks are described in relation to the following properties:

- inputRequired
- isDecimalDigit
- isHexDigit
- minimumInput
- needsSOSI
- validValues

The elementary checks precede use of the validator table (as described in relation to the **validatorDataTable** property), and all checks precede use of the **validatorFunction** property. This order of events is important because the validator function can do cross-field checking, and such checking often requires valid field values.

The value of **validatorFunction** is a validator function that you write. You code that function with no parameters and such that, if the function detects an error, it requests the re-display of the form by invoking `ConverseLib.validationFailed`.

If validation fails when you specify one of the two system functions, the displayed message is based on the value of the property **validatorFunctionMsgKey**. If validation fails when you specify a validator function of your own, however, the function does not use **validatorFunctionMsgKey**, but displays a message by invoking `ConverseLib.validationFailed`.

#### Related concepts

"Text forms" on page 188

#### Related reference

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## validatorFunctionMsgKey

The property **validatorFunctionMsgKey** (formerly the **validatorMsgKey** property) identifies a message that is displayed if the function specified in the property **validatorFunction** indicates an error.

In relation to text forms, these statements apply:

- The message is displayed in this case--
  - The **validatorFunction** property indicates use of `sysLib.verifyChkDigitMod10` or `sysLib.verifyChkDigitMod11`; and
  - The specified function indicates that the user's input is in error.
- The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.
- The value of **validatorFunctionMsgKey** is a string or literal that matches an entry of the first column in the message table.
- If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

#### Related concepts

"Text forms" on page 188

#### Related reference

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## validValues

The **validValues** property (formerly the **range** property) indicates a set of values that are valid for user input. The property is used for numeric or character fields. The format of the property is as follows:

```
validValues = arrayLiteral
```

### *arrayLiteral*

An array literal of singular and two-value elements, as in the following examples:

```
validValues = [ [1,3], 5, 12 ]  
validValues = [ "a", ["bbb", "i"]]
```

Each singular element contains a valid value. Each two-value element contains a range:

- For numbers, the leftmost value is the lowest that is valid, the rightmost is the highest. In the previous example, the values 1, 2, and 3 are valid for a field of type INT.
- For character fields, user input is compared against the range of values, for the number of characters for which a comparison is possible. For example, the range ["a", "c"] includes (as valid) any input whose first character is "a", "b", or "c". Although the string "cat" is greater than "c" in collating sequence, "cat" is valid input.

The general rule is as follows: if the first value in the range is called *lowValue* and the second is called *highValue*, the user's input is valid if *any* of these tests are fulfilled:

- User input is equal to *lowValue* or *highValue*
- User input is greater than *lowValue* and less than *highValue*
- The initial series of input characters matches the initial series of characters in *lowValue*, for as long as a comparison is possible
- The initial series of input characters matches the initial series of characters in *highValue*, for as long as a comparison is possible

Additional examples are as follows:

```
// valid values are 1, 2, 3, 5, 7, 9, and 11  
validValues = [[1, 3], 5, 7, 11]  
  
// valid values are the letters "a" and "z"  
validValues = ["a", "z"]  
  
// valid values are any string beginning with "a"  
validValues = ["a", "a"]  
  
// valid values are any string  
// beginning with a lowercase letter  
validValues = ["a", "z"]
```

If the user's input is outside the specified range, EGL runtime displays a message, as described in relation to the field property **validValuesMsgKey**.

### **Related concepts**

"Text forms" on page 188

### **Related reference**

"Validation properties" on page 68

"validationFailed()" on page 918

"DataTable part in EGL source format" on page 568

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## validValuesMsgKey

The property **validValuesMsgKey** (formerly the **rangeMsgKey** property) identifies the message that is displayed if the field property **validValues** is set and the user places out-of-range data into the field.

The *message table* (the table that contains the message) is identified in the program property **msgTablePrefix**. For details on the table name, see *DataTable part in EGL source format*.

The value of **validValuesMsgKey** is a string or literal that matches an entry of the first column in the message table.

If a numeric key is used with a message table that expects a character key, the number is converted to a character string. If a string literal is used with a message table that expects a numeric key, the value in the string must be a signed or unsigned integer.

This property applies only to numeric fields.

### Related concepts

"Text forms" on page 188

### Related reference

"DataTable part in EGL source format" on page 568

"validationFailed()" on page 918

"Validation properties" on page 68

"verifyChkDigitMod10()" on page 1043

"verifyChkDigitMod11()" on page 1044

## value

The property **value** identifies a string literal that is displayed as the field content when a Web page is displayed. That literal is used as a default when you place an EGL field on the Web Page in Page Designer.

### Related concepts

"Overview of EGL properties" on page 64

"PageHandler" on page 223

### Related tasks

"Associating an EGL record with a Faces JSP" on page 243

"Creating an EGL field and associating it with a Faces JSP" on page 242

"Creating an EGL pageHandler part" on page 221

"Using the Quick Edit view for PageHandler code" on page 244

### Related reference

"PageHandler field properties" on page 792

"PageHandler part properties" on page 788

"PageHandler part in EGL source format" on page 785

"Page Designer support for EGL" on page 221

## zeroFormat

The **zeroFormat** property specifies how zero values are displayed in numeric fields but not in fields of type MONEY. This property is affected by the **numeric separator**, **currency**, and **fillCharacter** properties. The values of **zeroFormat** are as follows:

### Yes

A zero value is displayed as the number zero, which can be expressed with decimal points (0.00 is an example, if the item is defined with two decimal places) and with currency symbols and character separators (\$000,000.00 is an example, depending on the values of the **currency** and **numericSeparator** properties). The following rules apply when the value of the property **zeroFormat** is *yes*:

- If the *fill character* (the value of the **fillCharacter** property) is 0, the data is formatted with the character 0
- If the fill character is a null, the data is left-justified
- If the fill character is a blank, the data is right-justified
- If the fill character is an asterisk (\*), asterisks are used as the left-side fillers instead of blanks

### No

A zero value is displayed as a series of the fill character.

### Related reference

"Java runtime properties (details)" on page 642

"Primitive field-level properties" on page 793

"set" on page 742

"currencySymbol" on page 471

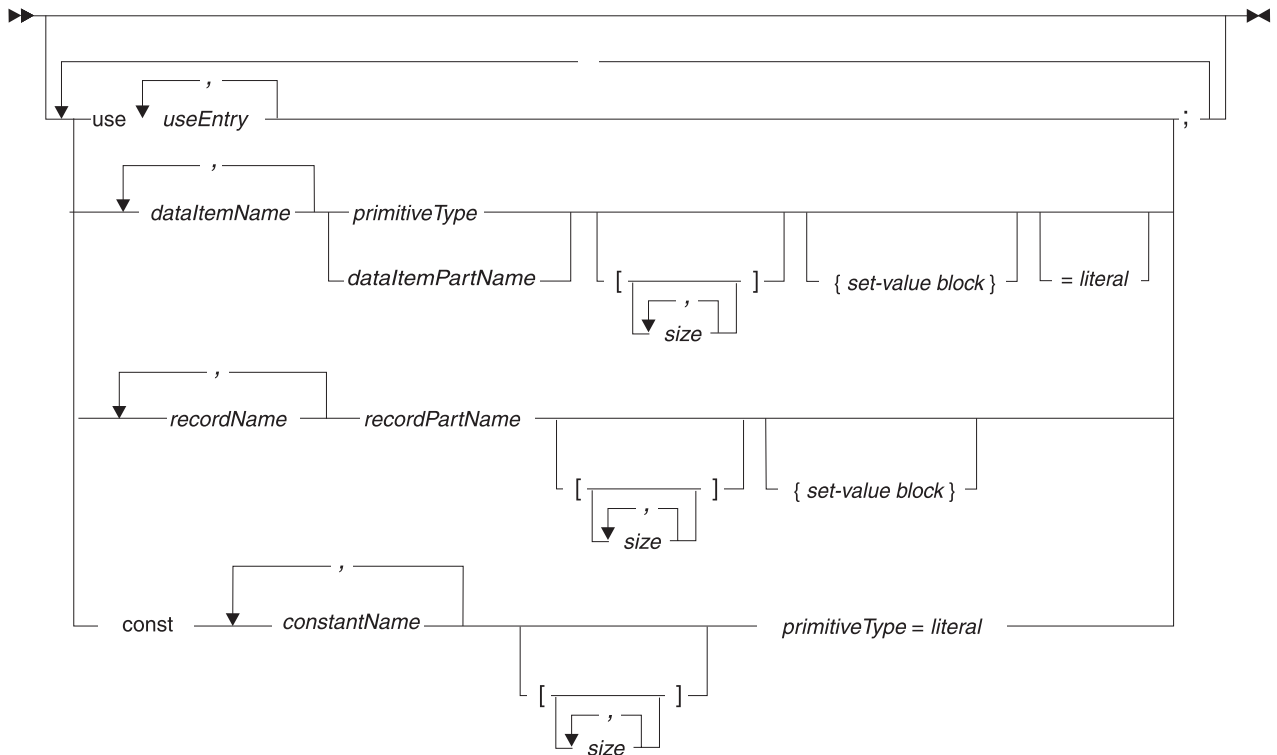
"Date, time, and timestamp format specifiers" on page 46

---

## Program data other than parameters

The syntax diagram for program data is as follows:





**use** *useEntry*

Provides easier access to a dataTable or library, and is needed to access to forms in a formGroup. For details, see *Use declaration*.

*dataItemName*

Name of a primitive field. For the rules of naming, see *Naming conventions*.

*primitiveType*

The type of a primitive field or (in relation to an array) the primitive type of an array element. Depending on the type, the following information may be required:

- The parameter's length or (in relation to an array), the length of an array element. The length is an integer that represents the number of characters or digits in the memory area.
- For some numeric types, you may specify an integer that represents the number of places after the decimal point. The decimal point is not stored with the data.
- For an item of type INTERVAL or TIMESTAMP, you may specify a datetime mask, which assigns a meaning (such as "year digit") to a given position in the item value.

For details, see *Primitive types* and the topic for a given type.

*dataItemPartName*

The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

*size*

Number of elements in the array. If you specify the number of elements, the array is initialized with that number of elements.

*set-value block*

For details, see *Overview of EGL properties* and *Set-value blocks*

*recordName*

Name of a record. For the rules of naming, see *Naming conventions*.

*recordPartName*

Name of a record part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

**const** *constantName primitiveType=literal*

Name, type, and value of a constant. Specify a quoted string (for a character type); a number (for a numeric type); or an array of appropriately typed values (for an array). Examples are as follows:

```
const myString String = "Great software!";  
const myArray BIN[] = [36, 49, 64];  
const myArray02 BIN[][] = [[1,2,3],[5,6,7]];
```

For the rules of naming, see *Naming conventions*.

### **Related concepts**

"EGL projects, packages, and files" on page 15

"Overview of EGL properties" on page 64

"Parts" on page 19

"Program part" on page 148

"References to variables in EGL" on page 59

"Segmentation in text applications" on page 189

"Set-value blocks" on page 68

"Syntax diagram for EGL statements and commands" on page 884

"Typedef" on page 28

### **Related reference**

"Arrays" on page 75

"Data initialization" on page 564

"DataItem part in EGL source format" on page 566

"DataTable part in EGL source format" on page 568

"EGL source format" on page 586

"EGL statements" on page 88

"forward" on page 686

"Function part in EGL source format" on page 621

"Indexed record part in EGL source format" on page 632

"Input form" on page 859

"Input record" on page 859

"INTERVAL" on page 42

"I/O error values" on page 638

"MQ record part in EGL source format" on page 769

"Naming conventions" on page 778

"Primitive types" on page 34

"Relative record part in EGL source format" on page 865

"Serial record part in EGL source format" on page 868

"SQL record part in EGL source format" on page 877

"TIMESTAMP" on page 44

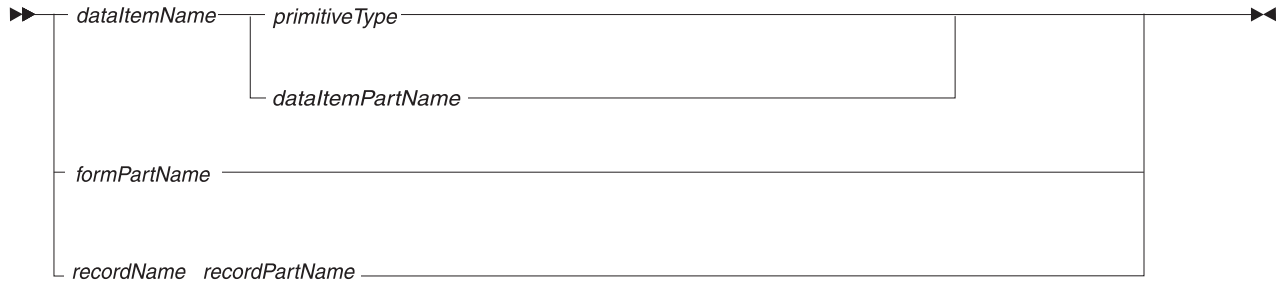
"VGUIRecord part in EGL source format" on page 1089

"Use declaration" on page 1091

---

## Program parameters

The syntax diagram for a program parameter is as follows:



### *dataItemName*

Name of a primitive field. For the rules of naming, see *Naming conventions*.

### *primitiveType*

The type of a primitive field. Depending on the type, the following information may be required:

- The parameter's length, which is an integer that represents the number of characters or digits in the memory area.
- For some numeric types, you may specify an integer that represents the number of places after the decimal point. The decimal point is not stored with the data.
- For an item of type INTERVAL or TIMESTAMP, you may specify a datetime mask, which assigns a meaning (such as "year digit") to a given position in the item value.

### *dataItemPartName*

The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

### *formPartName*

Name of a form.

The form must be accessible through a formGroup that is identified in one of the program's use declarations. A form accessed as a parameter cannot be displayed to the user, but can provide access to field values that are passed from another program.

For the rules of naming, see *Naming conventions*.

### *recordName*

Name of a record or fixed record. For the rules of naming, see *Naming conventions*.

### *recordPartName*

Name of a record part (or fixed-record part) that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

The following statements apply to input or output (I/O) against record parameters:

- A record passed from another program does not include record state such as the I/O error value *endOfFile*. Similarly, any change in the record state is not returned to the caller, so if you perform I/O against a record parameter, any tests on that record must occur before the program ends.
- Any I/O operation performed against the record uses the record properties specified for the parameter, not the record properties specified for the argument.
- For records of type *indexedRecord*, *mqRecord*, *relativeRecord*, or *serialRecord*, the file or message queue associated with the record declaration is treated as a run-unit resource rather than a program resource. Local record declarations share the same file (or queue) whenever the record property **fileName** (or **queueName**) has the same value. Only one physical file at a time can be associated with a file or queue name no matter how many records are associated with the file or queue in the run unit, and EGL enforces this rule by closing and reopening files as appropriate.

An arguments sent from another EGL program must be reference-compatible with the related parameter. For details, see *Reference compatibility in EGL*.

#### Related concepts

“Program part” on page 148

“References to parts” on page 23

“References to variables in EGL” on page 59

“Syntax diagram for EGL statements and commands” on page 884

“Typedef” on page 28

#### Related reference

“Arrays” on page 75

“Basic record part in EGL source format” on page 461

“DataItem part in EGL source format” on page 566

“EGL source format” on page 586

“Indexed record part in EGL source format” on page 632

“INTERVAL” on page 42

“Naming conventions” on page 778

“Primitive types” on page 34

“Reference compatibility in EGL” on page 862

“Relative record part in EGL source format” on page 865

“Serial record part in EGL source format” on page 868

“SQL record part in EGL source format” on page 877

“TIMESTAMP” on page 44

---

## Program part in EGL source format

You declare a program part in an EGL source file, which is described in *EGL source format*. When you write that file, do as follows:

- Include only those parts that are used exclusively by the program
- Do not include other generatable parts

The next example shows a called program part with two embedded functions, along with a standalone function and a standalone record part:

```
Program myProgram type basicProgram (employeeNum INT)
{
    includeReferencedFunctions = yes
}
```

```

// program-global variables
employees record_ws;
employeeName char(20);

// a required embedded function
Function main()
    // initialize employee names
    recd_init();

    // get the correct employee name
    // based on the employeeNum passed
    employeeName = getEmployeeName(employeeNum);
end

// another embedded function
Function recd_init()
    employees.name[1] = "Employee 1";
    employees.name[2] = "Employee 2";
end
end

// stand-alone function
Function getEmployeeName(employeeNum INT) returns (CHAR(20))

    // local variable
    index BIN(4);
    index = 2;
    if (employeeNum > index)
        return("Error");
    else
        return(employees.name[employeeNum]);
    end
end

// record part that acts as a typedef for employees
Record record_ws type basicRecord
    10 name CHAR(20)[2];
end

```

For other details, see the topic for a particular type of program.

### Related concepts

“Parts” on page 19

“Program part” on page 148

### Related reference

“Basic program in EGL source format”

“EGL source format” on page 586

“Function part in EGL source format” on page 621

“Program part properties” on page 856

“Text UI program in EGL source format” on page 844

“VGWebTransaction program in EGL source format” on page 847

## Basic program in EGL source format

An example of a basic program is as follows:

```

program myCalledProgram type basicProgram
    (buttonPressed int, returnMessage char(25))

function main()
    returnMessage = "";
    if (buttonPressed == 1)

```

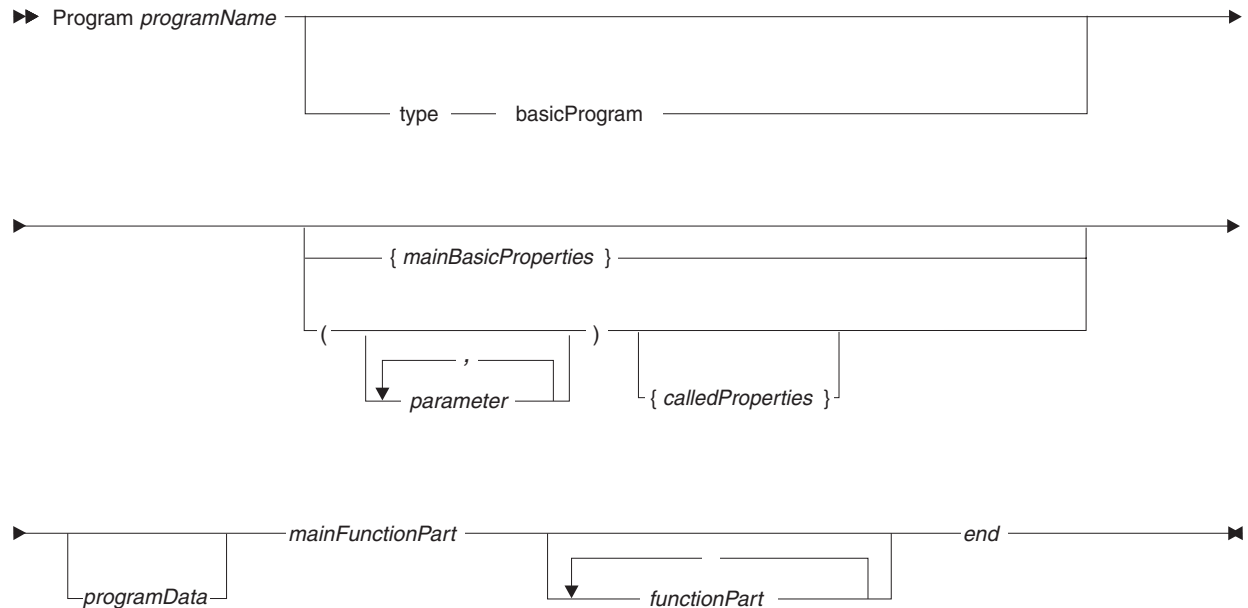
```

        returnMessage = "Message1";
    end

    if (buttonPressed == 2)
        returnMessage = "Message2";
    end
end
end

```

The syntax diagram for a program part of type `basicProgram` is as follows:



### Program *programPartName* ... end

Identifies the part as a program part and specifies the name and type. If the program name is followed by a left parenthesis, the program is a called basic program.

If you do not set the **alias** property (as described in *Program part properties*), the name of the generated program is *programPartName*.

For other rules, see *Naming conventions*.

### *mainBasicProperties*

The properties for a main basic program are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **inputRecord**
- **localSQLScope**
- **msgTablePrefix**
- **throwNrfEofExceptions**

For details, see *Program properties*.

#### *parameter*

Specifies the name of a parameter, which may be a data item, record, or form; or a dynamic array of records or data items. For rules, see *naming conventions*.

If the caller's argument is a variable (not a constant or literal), any changes to the parameter change the area of memory available to the caller.

Each parameter is separated from the next by a comma. For other details, see *Program parameters*.

#### *calledProperties*

The called properties are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **localSQLScope**
- **msgTablePrefix**
- **throwNrfEOFExceptions**

For details, see *Program properties*.

#### *programData*

Variable and use declarations, as described in *Program data other than parameters*.

#### *mainFunctionPart*

A required function named *main*, which takes no parameters. (The only program code that can take parameters is the program itself and functions other than *main*.)

For details on writing a function, see *Function part in EGL source format*.

#### *functionPart*

An embedded function, which is private to this program. For details on writing a function, see *Function part in EGL source format*.

### **Related concepts**

"EGL projects, packages, and files" on page 15

"Overview of EGL properties" on page 64

"Parts" on page 19

"Program part" on page 148

"Syntax diagram for EGL statements and commands" on page 884

### **Related reference**

"EGL source format" on page 586

"Function part in EGL source format" on page 621

"Naming conventions" on page 778

"Program data other than parameters" on page 837

"Program parameters" on page 840

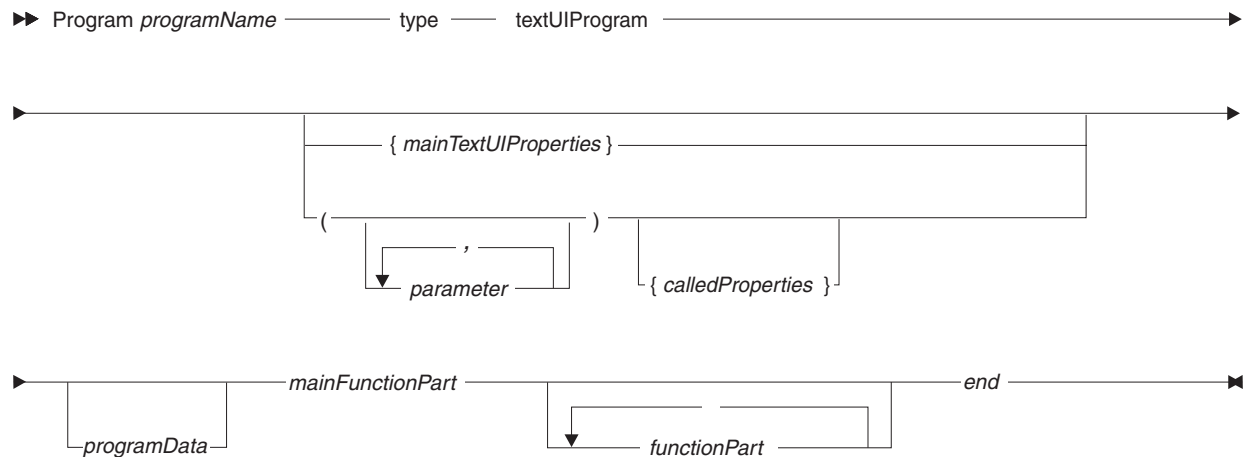
"Program part in EGL source format" on page 841

"Program part properties" on page 856

"Use declaration" on page 1091

## **Text UI program in EGL source format**

The syntax diagram for a program part of type `textUIProgram` is as follows:



### **Program** *programPartName* ... **end**

Identifies the part as a program part and specifies the name and type. If the program name is followed by a left parenthesis, the program is a called basic program.

If you do not set the **alias** property (as described later), the name of the generated program is *programPartName*. If you do not set the **alias** property (as described later), the name of the generated program is either *programPartName* or, if you are generating COBOL, the first eight characters of *programPartName*.

For other rules, see *Naming conventions*.

#### *mainTextUIProperties*

The properties for a main text UI program are optional:

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **inputForm**
- **inputRecord**
- **localSQLScope**
- **msgTablePrefix**
- **segmented**
- **throwNrfEofExceptions**

For details, see *Program properties*.

#### *parameter*

Specifies the name of a parameter, which may be a data item, record, or form; or a dynamic array of records or data items. For rules, see *naming conventions*.

If the caller's argument is a variable (not a constant or literal), any changes to the parameter change the area of memory available to the caller.

Each parameter is separated from the next by a comma. For other details, see *Program parameters*.

#### *calledProperties*

The called properties are optional:

- **alias**
- **allowUnqualifiedItemReferences**



- **includeReferencedFunctions**
- **msgTablePrefix**

For details, see *Program properties*.

#### *programData*

Variable and use declarations, as described in *Program data other than parameters*.

#### *mainFunctionPart*

A required function named *main*, which takes no parameters. (The only program code that can take parameters is the program itself and functions other than *main*.)

For details on writing a function, see *Function part in EGL source format*.

#### *functionPart*

An embedded function, which is not available to any logic part other than the program. For details on writing a function, see *Function part in EGL source format*.

An example of a Text UI program is as follows:

Program HelloWorld type textUIprogram

```
{
  use myFormgroup;
  myMessage char(25);

  function main()
    while (ConverseVar.eventKey not pf3)
      myTextForm.msgField = "          ";
      myTextForm.msgField="myMessage";
      converse myTextForm;
      if (ConverseVar.eventKey is pf3)
        exit program;
      end
      if (ConverseVar.eventKey is pf1)
        myMessage = "Hello Word";
      end
    end
  end
end
```

#### **Related concepts**

“EGL projects, packages, and files” on page 15

“Overview of EGL properties” on page 64

“Parts” on page 19

“Program part” on page 148

“Segmentation in text applications” on page 189

“Syntax diagram for EGL statements and commands” on page 884

#### **Related reference**

“EGL source format” on page 586

“Function part in EGL source format” on page 621

“Naming conventions” on page 778

“Program data other than parameters” on page 837

“Program parameters” on page 840

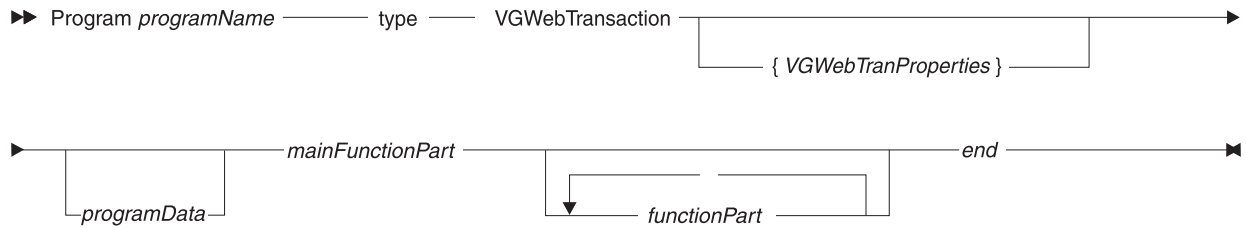
“Program part in EGL source format” on page 841

“Program part properties” on page 856

“Use declaration” on page 1091

## VGWebTransaction program in EGL source format

The syntax diagram is as follows for a program part of type VGWebTransaction:



### **Program** *programPartName* ... **end**

Identifies the part as a program part and specifies the name and type. A program of type VGWebTransaction cannot be called from another program.

If you do not set the **alias** property (as described later), the name of the generated program is *programPartName*.

For other rules, see *naming conventions*.

### *VGWebTranProperties*

The properties available for programs of type VGWebTransaction are as follows:

- **@DLI**
- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **inputRecord**
- **inputUIRecord**
- **includeReferencedFunctions**
- **msgTablePrefix**

For details on @DLI, see the topic of the same name. For other properties, see *Program part properties*.

### *programData*

Variable and use declarations, as described in *Program data other than parameters*.

### *mainFunctionPart*

A required function named *main*, which takes no parameters. (The only program code that can take parameters is the program itself, as well as functions other than *main*.)

For details on writing a function, see *Function part in EGL source format*.

### *functionPart*

An embedded function, which is not available to any logic part other than the program. For details on writing a function, see *Function part in EGL source format*.

### **Related concepts**

"EGL projects, packages, and files" on page 15

"Overview of EGL properties" on page 64

"Parts" on page 19

"Program part" on page 148

"Segmentation in text applications" on page 189

"Web support" on page 215

"Web transaction support in EGL" on page 163

#### Related reference

"@DLI" on page 322

"EGL source format" on page 586

"Function part in EGL source format" on page 621

"Naming conventions" on page 778

"Program data other than parameters" on page 837

"Program parameters" on page 840

"Program part in EGL source format" on page 841

"Program part properties" on page 856

"Syntax diagram for EGL statements and commands" on page 884

"Use declaration" on page 1091

### Gateway servlet parameters

By default, the gateway servlet has only one parameter defined: the parameter `hptGatewayProperties`. This parameter specifies a gateway properties file. The gateway properties file sets the remaining parameters for the gateway servlet. By default, the gateway properties file is the file `gw.properties` in the folder `JavaResources\JavaSource`. Comments in this file begin with the pound symbol (#).

Alternately, you can set the gateway servlet parameters in the same place as the `hptGatewayProperties` parameter is specified. However, it is better practice to set the parameters in the gateway properties file. The parameters in the gateway properties file override those you specify in the web application server.

Linkage properties are set in the linkage properties file. This file is specified in the `hptLinkageProperties` gateway servlet parameter. See *Linkage Properties*.

Table 11. Gateway Servlet Parameters

Servlet Parameter	Parameter Value	Notes
<code>hptDateMask</code>	<code>yyyy/dd/mm</code>	Required if any UI records have non-numeric date fields. For those fields, the datetime values that pass between tier 2 and tier 3 in either direction must correspond internally to a long Gregorian format for date and time. Parameter <code>hptDateMask</code> specifies that format. For web transactions generated as C++ programs, the format must match that specified in environment variable <code>EZERGRGL</code> . For web transactions generated as COBOL programs, the format in <code>hptDateMask</code> must match the datetime format specified during the server installation.

Table 11. Gateway Servlet Parameters (continued)

Servlet Parameter	Parameter Value	Notes
hptEntryApp	<i>web_transaction</i>	<p>Specifies a web transaction that provides an entry page. The program name must correspond to a name (such as a CICS definition) in the tier 3 environment. Also, an entry for the program name must be in the file to which parameter <i>hptLinkageProperties</i> refers.</p> <p>Either <i>hptEntryPage</i> or <i>hptEntryApp</i> is required. If both are specified, <i>hptEntryPage</i> is used.</p>
hptEntryPage	<i>/entry_page.jsp</i>	<p>Specifies the entry page JSP, relative to the document root.</p> <p>Either <i>hptEntryPage</i> or <i>hptEntryApp</i> is required. If both are specified, <i>hptEntryPage</i> is used.</p> <p>In the HTML output of the entry page JSP, the name of the web transaction invoked by the user must be assigned to <i>hptAppId</i>. Also, the SUBMIT button that invokes the web transaction must have the name <i>hptExec</i>, and the SUBMIT button that ends the web application server session must have the name <i>hptLogout</i>.</p> <p><i>Vagen1EntryPage.jsp</i> is provided as a working example. There, each web transaction name is included in a VALUE clause of a &lt;SELECT&gt; structure, and the NAME clause of that structure refers to <i>hptAppId</i>.</p>
hptErrorLog	<i>gateway_servlet_log_file</i>	<p>Optional. Specifies the fully qualified path of a log file. If this parameter is specified, the gateway servlet provides a trace of events and errors, and you can use the log to diagnose problems. Removing <i>hptErrorLog</i> turns off tracing.</p>

Table 11. Gateway Servlet Parameters (continued)

Servlet Parameter	Parameter Value	Notes
<code>hptExpiredPasswordPage</code>	<code>/expired_password_page.jsp</code>	<p>Optional. Specifies the expired password page JSP, relative to the document root.</p> <p>In the HTML output of the expired password page JSP, the <code>userid</code> input field must have the name <code>hptUserid</code>, the <code>old-password</code> input field must have the name <code>hptPassword</code>, the <code>new-password</code> input field must have the name <code>hptNewPassword</code>, the <code>new-password</code> input confirmation field must have the name <code>hptConfirmNewPassword</code>, and the <code>SUBMIT</code> button must have the name <code>hptExpiredPasswordPageLogin</code>.</p> <p><code>ExpiredPasswordPage.jsp</code> is provided as a working example of the JSP.</p> <p>The parameter <code>hptExpiredPasswordPage</code> has an effect only if <code>hptLogonCheck</code> is also specified.</p>
<code>hptGatewayProperties</code>	<code>gateway_properties_file</code>	Optional. Specifies the fully qualified path of a file that contains the other gateway servlet parameters.
<code>hptIDManageHost</code>	<code>host_name</code>	Optional. Specifies the TCP/IP host name for the machine where the Session ID Manager runs. The default is <code>localhost</code> .
<code>hptLinkageProperties</code>	<code>linkage_properties_file</code>	Required. Specifies the fully qualified name of the linkage properties file, which establishes a connection between the gateway servlet and each web transaction. See <i>Linkage properties</i> .

Table 11. Gateway Servlet Parameters (continued)

Servlet Parameter	Parameter Value	Notes
hptLogonCheck	<i>fully_qualified_class_name</i>	Optional. Specifies a logon-check class to handle user authentication.  RACFValidateLogin is provided as a working sample. To use that sample, specify the following:hptLogonCheck=com.ibm.hpt.gateway
hptLogonPage	<i>/logon_page.jsp</i>	Optional. Specifies the logon page JSP, relative to the document root. If this parameter is omitted or if hptPublicPassword and hptPublicUserid are present, no logon page is displayed.  In the HTML output of the logon page JSP, the userid input field must have the name <i>hptUserid</i> , the password input field must have the name <i>hptPassword</i> , and the SUBMIT button must have the name <i>hptLogin</i> .  Vagen1LogonPage.jsp is provided as a working example of the JSP.

Following is an example of a gateway properties file:

```
hptLogonPage=/Vagen1LogonPage.jsp
hptEntryPage=/Vagen1EntryPage.jsp
#hptEntryApp=WEBTXN1
hptErrorLog=c:/traces/Vagen1Gateway.log
hptLinkageProperties=c:/linktabs/csogwLinkage.properties
# use the following property if using Websphere 2.0,
# which uses JSP 0.91 and Servlet 2.0 support
#hptServletVersion=2.0
```

#### Related tasks

“Configuring a project to run Web transactions” on page 164

“Adding Web transaction support to an EGL Web project” on page 164

#### Related reference

“Linkage properties”

### Linkage properties

The linkage properties file tells the gateway servlet where to find the Web transactions and how to communicate with the Web transactions. The function provided by the file is similar to the function provided by the linkage table in VisualAge Generator Client/Server programs. This file is specified by the hptLinkageProperties parameter in the gateway servlet parameters. By default, the linkage properties are stored in the csogw.properties file in the folder JavaResources\JavaSource.

The linkage properties file contains three types of entries: application, serverLinkage, and options. Each application entry identifies one or more Web transactions and is related to serverLinkage entries that indicate how to connect to those Web transactions.

Following is an example of a linkage properties file:

```
application.WEBUITRAN=CICS5
application.WEBUI*=CICS5
application.STF*=idaho
application.MATT*=IMSC
application.BASIC*=rtpas400
application.Z*=remoteC

serverLinkage.CICS5.commtype=CICSECI
serverLinkage.CICS5.contable=CS0E037
serverLinkage.CICS5.location=nracics5
serverLinkage.CICS5.serverid=CPMI
serverLinkage.CICS5.javaProperty=my.pkg

serverLinkage.idaho.commtype=TCPIP
serverLinkage.idaho.contable=CS0X437
serverLinkage.idaho.location=machine01
serverLinkage.idaho.serverid=9877
serverLinkage.idaho.javaProperty=my.pak

serverLinkage.IMSC.commtype=TCPIMS
serverLinkage.IMSC.contable=CS0E037
serverLinkage.IMSC.location=carimsc
serverLinkage.IMSC.tcpport=4000
serverLinkage.IMSC.javaProperty=my.pak
serverLinkage.IMSC.msggroupid=mygroup
serverLinkage.IMSC.msdestid=IMSC
serverLinkage.IMSC.serverid=*

serverLinkage.rtpas400.commtype=as400
serverLinkage.rtpas400.contable=CS0E037
serverLinkage.rtpas400.location=rtpas400
serverLinkage.rtpas400.javaProperty=my.pag
serverLinkage.rtpas400.library=sdearth

serverLinkage.remoteC.commtype=cicsecl
serverLinkage.remoteC.contable=CS0I1252
serverLinkage.remoteC.location=CSONT2
serverLinkage.remoteC.serverid=CPMI
serverLinkage.remoteC.ctgLocation=ctghostname
serverLinkage.remoteC.ctgPort=2006
serverLinkage.remoteC.javaProperty=my.cicspkg
```

## Application entries

Each application entry has the following format:

```
application.webtran=servername
```

*webtran*

The name of the Web transaction.

The final character can be the wildcard character (\*). The entry `application.webui*=CICS5`, for example, is used for all Web transactions that start with the characters *webui*.

If multiple application entries match a Web transaction name, the most specific entry takes precedence. For example, a gateway servlet tries to access Web transaction *webuitran* when the linkage properties file contains the following entries:

```
application.webui*=abc application.webuit*=def
```

In this case, the second statement is used

#### *servername*

An arbitrary name that you assign to an application entry and to a related set of serverLinkage entries for a particular Web transaction. Consider the following application entry, for example:

```
application.SERVER1=CICS5
```

To access a Web transaction called *SERVER1*, the gateway servlet refers to the serverLinkage entries that are identified by *CICS5*.

## ServerLinkage entries

Each serverLinkage entry has the following format:

```
serverLinkage.servername.parameter=value
```

#### *servername*

The name of the Web transaction.

An arbitrary name that you assign to a set of serverLinkage entries for a particular Web transaction. The name must be present in an application entry.

#### *parameter*

One of the following:

##### **commtype**

Specifies the type of communications used to access the Web transaction. The valid values are as follows

**AS400** For use when the Web transaction resides on OS/400®.

##### **CICSECI**

For use when the Web transaction resides on CICS (CICS for AIX, CICS for MVS™, CICS for Solaris, or CICS for VSE).

##### **DIRECT**

For use when the Web transaction is a Java program, tier 2 and tier 3 are the same Windows 2000 or Windows NT machine, and you want the Web transaction to run in a thread of the Java Virtual Machine in which the gateway servlet is running.

##### **TCPIMS**

For use when the Web transaction resides on an IMS system.

**TCPIP** For use when the Web transaction resides on a native Windows 2000, Windows NT, OS/2, AIX, HP-UX or Solaris system. If the tier 2 and tier 3 environments are the same Windows 2000 or Windows NT machine, consider using commtype **DIRECT** instead of **TCPIP**.

##### **contable**

Specifies the conversion table used on the tier 2 platform. The format is as follows:

```
CS0zxxxx;
```

**z** One of the following binary formats, which refers to the tier 3 platform:

- I (for Intel)
- E (for EBCDIC)



- J (for Java Unicode)
- X (for UNIX)

**xxxx** The code page used for conversion.

The SUN Java conversion routines convert the data in accordance with the data definitions in the UI record. For more information on conversion tables for different languages and platforms, see the *VisualAge Generator Client/Server Communications Guide*.

### **ctgLocation**

Optional. Specifies the machine where the CICS Transaction Gateway resides. If you do not specify **ctgLocation** and **ctgPort** when **commtype=CICSECI**, the CICS Transaction Gateway is assumed to be local.

### **ctgPort**

Optional. Specifies the port on which where the remote CICS Transaction Gateway listens. If you do not specify **ctgLocation** and **ctgPort** when **commtype=CICSECI**, the CICS Transaction Gateway is assumed to be local.

**Note:** The procedure for setting the port number of the CICS Transaction Gateway listener depends on the version of that product: for version 3.01, use the **ctgStart -port** command; for version 3.03, use the **JGate -port** command; for other versions, see the CICS Transaction Gateway configuration manual. The default port number is 2006.

### **location**

For **commtype=CICSECI**, specifies the CICS system identifier that corresponds to the server name in the **CICSCLI.INI** file of the CICS Client. For **commtype=TCPIMS**, **commtype=TCPIP**, or **commtype=AS400**, specifies the TCP/IP host name of the machine where the Web transaction resides.

### **remoteapptype**

If the Web transaction is a Java program and **commtype=TCPIP**, specify the value **VGJAVA**; otherwise, do not include this parameter.

### **serverid**

For **commtype=CICSECI**, specifies the CICS trans-id for the catcher. In most cases, the following is true:

- The value of *serverid* is **CPMI**, which causes invocation of program **DFHMIRS**.
- If you specify a trans-id other than **CPMI**, CICS starts **CPMI**, which switches control to the CICS transaction you specify, which in turn switches control to program **DFHMIRS**.

To avoid starting **CPMI** when the trans-id is not **CPMI**, prepend **tpn\_** to the **trans\_id**. If the trans-id is **WEBT**, for example, specify **tpn\_WEBT**; but if the trans-id is **CPMI**, specify only **CPMI**. If you specify **WEBT** without **tpn\_**, CICS starts **CPMI**, which in turn switches control to **WEBT**.

**Note:** In relation to OS/390®, *serverid* is ignored and the CICS transaction always runs as **CPMI** unless the following is true:

- CICS TS V1.3 is installed with PTF **UQ47399**.

- CICS Transaction Gateway is at or above V3.1.2.

For `commtype=TCPIP`, *serverid* specifies the port number of the listening socket on the machine where the Web transaction resides.

For `commtype=TCPIMS`, *serverid* specifies the IMS transaction code. If you set `serverid=*`, the value of *servername* is used as the transaction code.

For `commtype=AS/400`, *serverid* is not used.

#### **tcpport**

For `commtype=TCPIMS`, specifies the port number of the listening socket on the machine where the Web transaction resides. The listening socket is configured in the IMS TCPIP Open Transaction Manager Access connection (ITOC).

#### **javaProperty**

Specifies the Java package where the UI record object and UI record bean for the Web transaction are located. This entry is case sensitive.

#### **imgroupid**

Specifies a RACF<sup>®</sup> group to which the user must be connected for authentication. This entry is valid only for TCPIMS connections.

#### **imsdestid**

Specifies the IMS system in which the Web transaction runs, as that system is defined by the ITOC configuration. This entry is valid only for TCPIMS connections.

#### **library**

Specifies the name of the library on the OS/400 system where the Web transaction resides. If the tier 3 platform is OS/400 and you omit this entry or leave it blank, VisualAge Generator searches for the Web transaction in the QVGEN library, then in the library list specified by OS/400 variable QUSRLIBL.

#### *value*

The value to which the parameter is set.

## **Options entries**

The following options entries are valid:

- `hptGateway.propertiesRefreshInterval=n`

*n* Specifies the number of minutes that pass between gateway-servlet inspections of the linkage properties file.

This entry lets you dynamically put into effect any changes made to the linkage properties file, without your being required to stop and start the Web application server. Web transactions initiated after the inspection are invoked with the new values. A value of 0 indicates that the gateway servlet reviews the linkage properties file only at startup.

- `application.webtran.traceFlag=n`

#### *webtran*

The name of the Web transaction, the same as the same-named parameter in application entries.

*n* Indicates whether the gateway servlet is to provide internal, communications layer tracing, which is useful only if you are in contact with IBM support. Alternatives are 1 (for tracing) or 0 (for none, as is

the default). A communications layer trace requires the presence of the gateway servlet parameter `hptErrorLog`, which is usually in the gateway properties file.

#### Related tasks

“Configuring a project to run Web transactions” on page 164

“Adding Web transaction support to an EGL Web project” on page 164

#### Related reference

“Gateway servlet parameters” on page 848

---

## Program part properties

Program part properties vary by whether the program is called or main and, if main, by whether the program is of type basic, textUI, or VGWebTransaction. The properties are as follows:

#### @DLI

This complex property lets you specify the call interface, PSB, and PCBs for an EGL program that uses a DL/I database. For more information and details about the fields you can change in this property, see *@DLI*.

#### alias = "alias"

A string that is incorporated into the names of generated output. If you do not set the **alias** property, the program-part name is used instead.

The **alias** property is available in any program.

#### allowUnqualifiedItemReferences = no, allowUnqualifiedItemReferences = yes

Specifies whether to allow your code to omit container and substructure qualifiers when referencing items in structures.

The **allowUnqualifiedItemReferences** property is available in any program.

Consider the following record part, for example:

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

The following variable is based on that part:

```
myRecord aRecordPart;
```

If you accept the default value of **allowUnqualifiedItemReferences** (*no*), you must specify the record name when referring to `myItem01`, as in this assignment:

```
myValue = myRecord.myItem01;
```

If you set the property **allowUnqualifiedItemReferences** to *yes*, however, you can avoid specifying the record name:

```
myValue = myItem01;
```

It is recommended that you accept the default value, which promotes a best practice. By specifying the container name, you reduces ambiguity for people who read your code and for EGL.

EGL uses a set of rules to determine the area of memory to which a variable name or item name refers. For details, see *References to variables and constants*.

**handleHardIOErrors = yes, handleHardIOErrors = no**

Sets the default value for the system variable **VGVar.handleHardIOErrors**. The variable controls whether a program continues to run after a hard error has occurred on an I/O operation in a **try** block. The default value for the property is *yes*, which sets the variable to 1.

Code that was migrated from VisualAge Generator may not work as before unless you set **handleHardIOErrors** to *no*, which sets the variable to 0.

This property is available in any program. For other details, see *VGVar.handleHardIOErrors* and *Exception handling*.

**includeReferencedFunctions = no, includeReferencedFunctions = yes**

Indicates whether the program contains a copy of each function that is neither inside the program nor in a library accessed by the program.

The **includeReferencedFunctions** property is available in any program.

The default value is *no*, which means that you can ignore this property if you are fulfilling the following practices at development time, as is recommended:

- Place shared functions in a library
- Place non-shared functions in the program

If you are using shared functions that are not in a library, generation is possible only if you set the property **includeReferencedFunctions** to *yes*.

**inputForm = "formName"**

Identifies a form that is presented to the user before the program logic runs, as described in *Input form*.

The **inputForm** property is available only in main text UI programs.

**inputUIRecord = VGUIRecord**

Identifies a VGUI record that a VGWebTransaction program receives from a user's Web-page submission.

Two cases apply:

- The Web page was presented by a **show** statement with a **returning** clause; and the user submitted the Web page to fulfill the intent of that statement.
- Alternatively, a **show** statement with a **returning** clause was not involved; nevertheless, the user clicked a Web-page button or hypertext link to invoke an EGL program and to submit VGUI record data to that program.

The name of a given field in the VGUI record on the Web page must have the same name and type as the corresponding field in the record identified by the property **inputUIRecord**.

Data is not placed in the **inputUIRecord** property in these cases:

- A **show** statement invokes the program with a set of arguments instead of with a VGUI record; or
- The Web page was presented by a **converse** statement; and the user submitted the Web page to return to the next statement in the same program.

The **inputUIRecord** property is available only in VGWebTransaction programs.

For additional details, see *show*.

**inputRecord** = *"inputRecord"*

Identifies a global, basic record that a program automatically initializes and that may receive data from a program that uses a **transfer** statement to transfer control. For additional details, see *Input record*.

The **inputRecord** property is available in any main program.

**localSQLScope** = **yes**, **localSQLScope** = **no**

Indicates whether identifiers for SQL result sets and prepared statements are local to the program, as is the default. If you accept the value *yes*, different programs can use the same identifiers independently.

If you specify *no*, the identifiers are shared throughout the run unit. The identifiers created in the current code are available elsewhere, although other code can use **localSQLScope** = **yes** to block access to those identifiers. Also, the current code may reference identifiers created elsewhere, but only if the other code was already run and did not block access.

The effects of sharing SQL identifiers are as follows:

- You can open a result set in one program and get rows from that set in another
- You can prepare an SQL statement in one program and run that statement in another

The **localSQLScope** property is available in any program.

**msgTablePrefix** = *"prefix"*

Specifies the first one to the four characters in the name of the DataTable that is used as the message table for the program. The other characters in the name correspond to one of the national language codes listed in *DataTable part in EGL source format*.

The **msgTablePrefix** property is available in any basic, textUI, or VGWebTransaction program. For other details on messages that are used in a VGWebTransaction program, see the topic for build descriptor option **msgTablePrefix**.

PageHandlers do not use a message table, but use a JavaServer Faces message resource. For details on that resource, see the description of the **msgResource** property in *PageHandler part in EGL source format*,

**segmented** = **no**, **segmented** = **yes**

Indicates whether the program is segmented, as explained in *Segmentation*. The value is always set to *yes* in VGWebTransaction programs, and the default is *no* in main text UI programs. The property is not valid in other types of programs.

**throwNrfEofExceptions** = **no**, **throwNrfEofExceptions** = **yes**

Specifies whether a soft error causes an exception to be thrown. The default is *no*. For background information, see *Exception handling*.

#### Related concepts

"Program part" on page 148

"References to variables in EGL" on page 59

"Segmentation in text applications" on page 189

#### Related reference

"@DLI" on page 322

"DataTable part in EGL source format" on page 568

- "Exception handling" on page 94
- "forward" on page 686
- "handleHardIOErrors" on page 1082
- "Input form"
- "Input record"
- "msgTablePrefix" on page 484
- "Naming conventions" on page 778
- "PageHandler part in EGL source format" on page 785
- "show" on page 751
- "Syntax diagram for EGL statements and commands" on page 884
- "VGUIRecord part in EGL source format" on page 1089

## Input form

When you declare a main program that runs in a text application, you have the option to specify an *input form*, which is a form that is presented to the user before the program logic runs.

Two scenarios are possible:

- If the program is the target of a *show-form-returning-to* statement from an EGL-generated program, the sending program presents a form to the user, and that form must be identical to the input form of the receiving program. The receiving program is invoked only after the user submits the form. After the user submits the form, the receiving program does not present the input form a second time; instead, the initial logic (the *execute* function) runs.
- If the program is the target of a *transfer* statement from a program (EGL or non-EGL) or if the program is invoked by the user or by an operating-system command, the receiving program converses the input form. (In this case, input fields on that form are initialized before display.) After the user submits the form, the initial logic (the *execute* function) runs.

The input form must be in the form group that you specified in the program-part declaration.

### Related reference

"Data initialization" on page 564

## Input record

Any main program part can have an input record, which is a global record that the EGL-generated program automatically initializes. The record must be of type `basicRecord`.

If the program starts as a result of a *transfer* with a record, the program initializes the input record (which is internal to that program), then assigns the transferred data to the record.

If the input record is longer than the received data, the extra area in the input record retains the values assigned during record initialization. If the input record is shorter than the received data, the extra data is truncated.

If primitive types in the transferred data are incompatible with the primitive types in the equivalent positions in the input record, the receiving program may end abnormally.

**Related concepts**

"Overview of EGL properties" on page 64

"Parts" on page 19

"Compatibility with VisualAge Generator" on page 532

**Related reference**

"Data initialization" on page 564

---

## Record and file type cross-reference

The next table shows the association of record type and file type, by target platform.

**Related concepts**

"Record types and properties" on page 138

"Resource associations and file types" on page 393

**Related task**

"Adding a resource associations part to an EGL build file" on page 397

"Editing a resource associations part in an EGL build file" on page 397

"Removing a resource associations part from an EGL build file" on page 398

**Related reference**

"resourceAssociations" on page 486

---

## Properties that support variable-length records

When you declare a record part, you can include properties that support the use of variable-length records. You can use variable-length serial records for accessing sequential files, variable-length serial or indexed records for accessing VSAM files, and variable-length MQ records for accessing MQSeries message queues.

### Variable-length records with the `lengthItem` property

The `lengthItem` property, if present, identifies an item that is used when:

- Your code reads a record from a file or queue. The length item receives the number of bytes read into the variable-length record.
- Your code writes a record. The length item specifies the number of bytes to add to the file or queue.

The length item can be any of the following:

- A structure item in the same record
- A structure item in a record that is global to the program or is local to the function that accesses the record (the length item may be qualified with a record variable declared in the program or function)
- A data item that is global to the program or is local to the function that accesses the record

The length item has these characteristics:

- Has a primitive type of BIN, DECIMAL, INT, NUM, or SMALLINT
- Contains no decimal place
- Allows for 9 digits at most



An example of a variable-length record part with the `lengthItem` property is as follows:

```
Record mySerialRecordPart1 type serialRecord
{
  fileName = "myFile",
  lengthItem = "myOtherField"
}
10 myField01 BIN(4);    // 2 bytes long
10 myField02 NUM(3);    // 3 bytes long
10 myField03 CHAR(20); // 20 bytes long
end
```

When writing a record, the value of the length item must fall between item boundaries, unless the item is a character item. For example, a record of type `mySerialRecordPart1` can have the length item, `myOtherField`, set to 2, 5, 6, 7, ... , 24 , 25. A record with `myOtherField` set to 2 only contains a value for `myField01`; a record with `myOtherField` set to 5 contains values for `myField01` and `myField02`; a record with `myOtherField` set to 6 through 24 also contains part of `myField03`.

## Variable-length records with the `numElementsItem` property

The *NumElementsItem* property, if present, identifies an item that is used when your code adds to or updates the file or queue. The variable-length record must have an array as the last, top-level structure item. The value in the number of elements item represents the actual number of array elements that are written. The value can range from 0 to the maximum, which is the *occurs* value specified in the declaration of the last, top-level structure item in the record.

The number of bytes written is equal to the sum of the following:

- The number of bytes in the fixed-length part of the record.
- The value of the number of elements item multiplied by the number of bytes in each element of the ending array.

The number of elements item has these characteristics:

- Has a primitive type of BIN, DECIMAL, INT, NUM, SMALLINT
- Contains no decimal place
- Allows for 9 digits at most

An example of a variable-length record part with the `numElementsItem` property is as follows:

```
Record mySerialRecordPart2 type serialRecord
{
  fileName = "myFile",
  numElementsItem = "myField02"
}
10 myField01 BIN(4);    // 2 bytes long
10 myField02 NUM(3);    // 3 bytes long
10 myField03 CHAR(20)[3]; // 60 bytes long
  20 mySubField01 CHAR(10);
  20 mySubField02 CHAR(10);
end
```

Writing a record of type `mySerialRecordPart2` with the number of elements item `myField02` set to 2 results in a variable-length record with `myField01`, `myField02`, and two occurrences of `myField03` being written to the file or queue.



The number of elements item must be an item in the fixed-length part of the variable-length record. Use an unqualified reference to name the number of elements item. For example, use `myField02` rather than `myRecord.myField02`.

The number of elements item has no effect when you are reading a record from the file.

## Variable-length records with both `lengthItem` and `numElementsItem` properties

If both the `lengthItem` and the `numElementsItem` properties are specified for a variable-length record, the length of the record is calculated using the number of elements item. The calculated length is moved to the record length item before the record is written to the file.

## Variable-length records passed on a call or transfer

If variable-length records are passed on a call, these statements apply:

- Space is reserved for the maximum length specified for the record
- If the value of the `callLink` element, property **type**, is `remoteCall` or `ejbCall`, the length item (if any) must be inside the record; for details, see *callLink element*

Similarly, if variable-length records are passed on a transfer, space is reserved for the maximum length specified for the record.

### Related concepts

“MQSeries support” on page 336

“Record types and properties” on page 138

### Related reference

“callLink element” on page 499

“MQ record properties” on page 772

---

## Reference compatibility in EGL

Reference-compatibility rules apply to reference and non-reference variables, as described in the next sections.

### For reference variables

In relation to reference variables, the following statements apply:

- In an assignment statement, you can assign a reference variable only to a reference variable that is based on the same part
- In a function invocation, you can pass a reference variable only to a parameter that is based on the same part
- You can return a reference variable from a function, but the return field (which may be a value in another function invocation) must be based on the same part

A part that is defined in one EGL package is not the same as a part defined in another package, even if the part names are identical.

For an overview of reference variables, see *Reference variables and NIL in EGL*.

## For non-reference variables

A set of reference-compatibility rules are in effect when EGL transfers a non-reference value between an argument and the related parameter, but only in the following situation:

- The parameter in the receiving function has the modifier INOUT; or
- The parameter is in the onPageLoad function of a PageHandler.

In relation to a fixed record or structure field, the length of the argument must be greater than or equal to the length of the parameter. This rule prevents the receiving code from accessing memory that is not valid.

Also, when the argument is referring to a primitive type or an array of DataItems, the following statements apply:

- The primitive characteristics (if any) must be identical. For example, an argument of type CHAR(6) is not compatible with a parameter of type CHAR(7). When you pass literals, however, EGL truncates and pads values, as described in *Assignment compatibility in EGL*.
- An argument that is nullable is compatible with a nullable or non-nullable parameter. An argument that is not nullable is compatible only with a non-nullable parameter.

### Related concepts

"PageHandler" on page 223

### Related reference

"Function parameters" on page 616

"Function part in EGL source format" on page 621

"PageHandler part in EGL source format" on page 785

"Program parameters" on page 840

"Program part in EGL source format" on page 841

"Reference variables and NIL in EGL"

---

## Reference variables and NIL in EGL

A parameter or variable is an area of memory. In some cases, the variable contains the business data of interest; a particular name or employee ID, for example. In other cases, the variable is a *reference variable*, which contains a memory address that is used to access the business data at run time.

A reference variable in EGL is based on one of the following parts, each of which is also called a *reference type*:

- ConsoleField
- Interface
- Menu
- MenuItem
- Prompt
- Report
- ReportData
- Service

Except for Service and Interface parts, you do not define any of the previously mentioned parts, which are predefined by EGL.

Implications are as follows:

### Initialization

When you declare a reference variable, EGL usually initializes the variable to the value `NIL`, which indicates that the variable does not point to any business data. In the following cases, the value `NIL` is not the initial value:

- The variable is based on a part that includes a field initialized at definition time; or
- The variable declaration includes a set-value block.

In the cases just mentioned, the variable refers to an area of memory that is created when the variable is declared.

When you declare an array of reference variables (for example, `MenuItem[3]`), the preceding rules also apply to each element in the array.

### In assignment statements

After you assign a reference variable to another reference variable, the source and target each contain a value that is used to access the same area of memory. You can assign a reference variable only to another reference variable that is based on the same part, and in the case of an interface or service part, a part defined in one package is not the same as a part defined in another package, even if the part names are identical.

**Note:** When you assign a non-reference variable to another non-reference variable, the situation is different:

- The source and target each contain a copy of the same business data. If a non-reference source variable in an assignment statement contains a specific employee ID, for example, the statement causes the target variable to contain that ID as well.
- The variables may be of different types, as described in *Assignment compatibility*.

### In comparisons

The operator that tests for equality (`==`) and inequality (`!=`) are valid for comparing reference variables. The comparison is based on the area of memory to which the variables refer. If two variables refer to different areas of memory, the two variables are not equal even if the business data is identical.

### In function invocations

You can pass a reference variable only to a parameter of the same type; and in the case of an interface or service part, a part defined in one package is not the same as a part defined in another package, even if the part names are identical. However, you can pass a reference variable to a service function only if the parameter is of the same type as the argument and only if the invocation is local; specifically, only if *local* is the value of the complex property `@EGLBinding`, field `commType`.

The function always receives a copy of the variable being passed:

- Assigning a different value to the data referenced by the parameter (for example, changing the color of a console field) changes the data that is available in the invoking function
- However, using the parameter on the left side of an assignment statement in the invoked function has no effect on the invoking function. After the assignment, the parameter refers to the same area of memory as the area referenced by the variable on the right side of the assignment statement, but the argument in the invoking function is unaffected by that assignment.

The parameter modifiers IN, OUT, and INOUT are not valid for parameters of a reference type.

Any reference type is valid as a return type, which means that a copy of an address is returned to the invoking function at run time. Even if the reference type is declared as a local variable in the invoked function, the business data (and an area of memory to hold that data) is retained in the invoking function.

#### In calls to programs

You cannot pass a reference variable to another program.

#### In PageHandlers

You cannot pass a reference variable to the onPageLoad function in a PageHandler; nor can you place a reference variable in (or retrieve a reference variable from) the session or request object.

#### Related concepts

#### Related reference

"Reference compatibility in EGL" on page 862

---

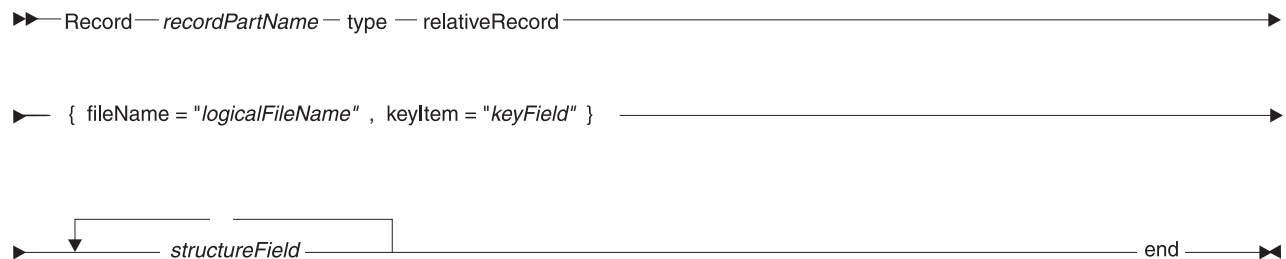
## Relative record part in EGL source format

You declare a fixed record part of type relativeRecord in an EGL source file, which is described in *EGL source format*.

An example of a relative record part is as follows:

```
Record myRelativeRecordPart type relativeRecord
{
    fileName = "myFile",
    keyItem  = "myKeyItem"
}
10 myKeyItem NUM(4);
10 myContent CHAR(76);
end
```

The syntax diagram of a relative record part is as follows:



#### Record *recordPartName* **relativeRecord**

Identifies the part being of type *relativeRecord* and specifies the name. For the rules of naming, see *Naming conventions*.

#### **fileName** = *"logicalFileName"*

The logical file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

#### **keyItem** = *"keyField"*

The key field, which can be any of these areas of memory:

- A field in the same fixed record

- A variable or field that is global to the program or is local to the function that accesses the fixed record

You must use an unqualified reference to name the key field. For example, use *myField* rather than *myRecord.myField*. (In a function, however, you can reference the key field as you would reference any field.) The key field must be unique in the local scope of the function that accesses the record or must be absent from local scope and unique in global scope.

The key field has these characteristics:

- Has a primitive type of NUM, BIN, DECIMAL, INT, or SMALLINT
- Contains no decimal place
- Allows for 9 digits at most

Only the **get** and **add** statements use the relative record key field, but the key field must be available to any function that uses the fixed record for file access.

#### *structureField*

A structure field, as described in *Structure field in EGL source format*.

#### **Related concepts**

“EGL projects, packages, and files” on page 15  
 “References to parts” on page 23  
 “Parts” on page 19  
 “Record parts” on page 135  
 “References to variables in EGL” on page 59  
 “Typedef” on page 28

#### **Related tasks**

“Syntax diagram for EGL statements and commands” on page 884

#### **Related reference**

“Arrays” on page 75  
 “DataItem part in EGL source format” on page 566  
 “EGL source format” on page 586  
 “Function part in EGL source format” on page 621  
 “Indexed record part in EGL source format” on page 632  
 “MQ record part in EGL source format” on page 769  
 “Naming conventions” on page 778  
 “Primitive types” on page 34  
 “Program part in EGL source format” on page 841  
 “Resource associations and file types” on page 393  
 “Serial record part in EGL source format” on page 868  
 “SQL record part in EGL source format” on page 877  
 “Structure field in EGL source format” on page 880

---

## **Run unit**

A *run unit* is a set of programs that are related by local calls or (in some cases) by transfers. Each run unit has these characteristics:

- The programs operate together as a group. When a hard error occurs but is not handled, all the programs in the run unit are removed from memory.
- The programs share the same runtime properties. The same databases and files are available throughout the run unit, for example, and when you invoke

sysLib.connect or VGLib.connectionService to connect to a database dynamically, the connection is present in any program that receives control in the same run unit.

The *Java run unit* is composed of programs that run in a single thread. A new run unit can start with a main program, as when the user invokes the program. A **transfer** statement also invokes a main program but continues the same run unit.

In the following cases, a called program is the initial program of a run unit:

- The call is a call from an EJB session bean; or
- The call is a remote call, except that the same run unit continues in the following case--
  - The called program is generated by EGL or VisualAge Generator; and
  - No TCP/IP listener is involved in the call.

All programs in a Java run unit are affected by the same Java runtime properties.

#### **Related concepts**

"Java runtime properties" on page 431

"Linkage options part" on page 399

#### **Related reference**

"Default database" on page 298

"connect()" on page 1025

"connectionService()" on page 1047

---

## **resultSetID**

The result-set identifier is in the EGL syntax and is used when you are accessing a relational database and need to relate the following kinds of statements:

- First, an **open** or **get** statement that selects a result set, or an **open** statement that calls a stored procedure that returns a result set
- Second, the statements that access the result set

If you are using an SQL record as the I/O object, the record name is sufficient to relate one kind of statement to another, unless you modify the SQL statements associated with the record to retrieve different sets of columns for update in different statements. In this case, use a result-set identifier to identify the result set associated with an EGL **replace** statement.

#### **Related concepts**

"SQL support" on page 277

#### **Related reference**

"replace" on page 738

"open" on page 722

"get" on page 687

---

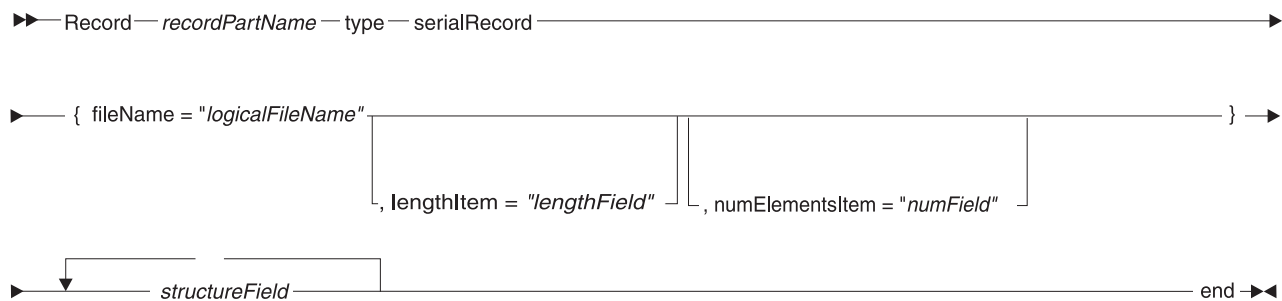
## Serial record part in EGL source format

You declare a record part of type `serialRecord` in an EGL source file, which is described in *EGL source format*.

An example of a serial record part is as follows:

```
Record mySerialRecordPart type serialRecord
{
    fileName = "myFile"
}
10 myField01 CHAR(2);
10 myField02 CHAR(78);
end
```

The syntax diagram for a serial record part is as follows:



### Record *recordPartName* serialRecord

Identifies the part as being of type `serialRecord` and specifies the part name. For the rules of naming, see *Naming conventions*.

### **fileName** = "logicalFileName"

The logical file name. For details on the meaning of your input, see *Resource associations (overview)*. For rules, see *Naming conventions*.

### **lengthItem** = "lengthField"

The length field, as described in *Properties that support variable-length records*.

### **numElementsItem** = "numField"

The number of elements field, as described in *Properties that support variable-length records*.

### *structureField*

A structure field, as described in *Structure field in EGL source format*.

### Related concepts

"EGL projects, packages, and files" on page 15

"References to parts" on page 23

"Parts" on page 19

"Record parts" on page 135

"References to variables in EGL" on page 59

"Resource associations and file types" on page 393

"Typedef" on page 28

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

**Related reference**

"Arrays" on page 75  
"DataItem part in EGL source format" on page 566  
"EGL source format" on page 586  
"Function part in EGL source format" on page 621  
"Indexed record part in EGL source format" on page 632  
"MQ record part in EGL source format" on page 769  
"Naming conventions" on page 778  
"Primitive types" on page 34  
"Program part in EGL source format" on page 841  
"Properties that support variable-length records" on page 860  
"Relative record part in EGL source format" on page 865  
"SQL record part in EGL source format" on page 877  
"Structure field in EGL source format" on page 880

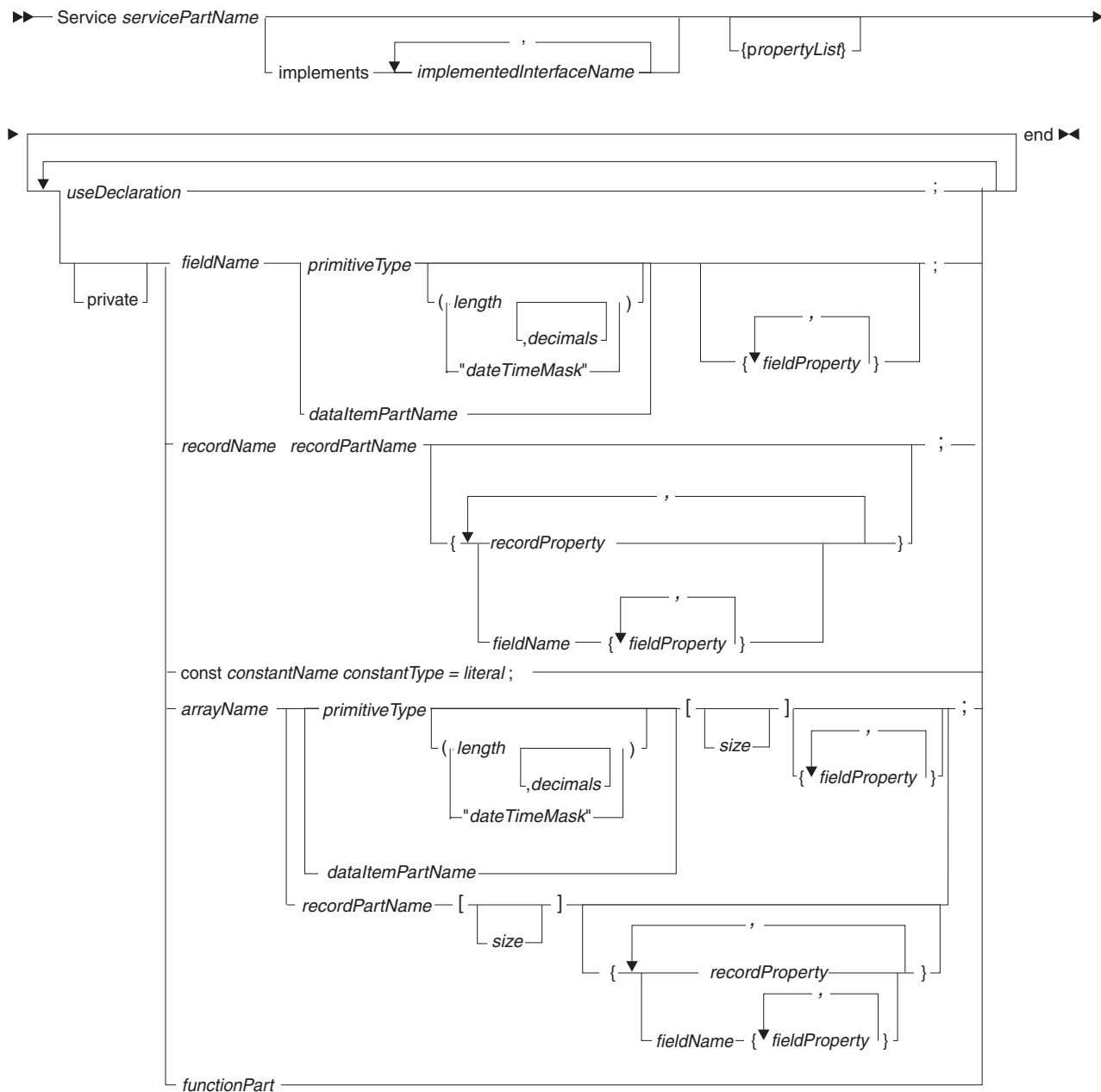
---

## Service part in EGL source format

For an overview of services, see *EGL services*. You can declare services in an EGL file, which is described in *EGL source format*.

The syntax diagram for a Service part is here:





### **Service** *servicePartName* ... **end**

Identifies the part as a service and specifies the part name. For the rules of naming, see *Naming conventions*.

### **implements***interfaceName*

Indicates that the service implements the specified interfaces. The primary meaning is that the service contains every function described in the interface. These statements also apply:

- In an assignment statement, a variable of the interface type can receive a variable of the service type:  

```
myInterface = myService;
```
- Similarly, in a function invocation, a parameter of the interface type can receive an argument of the service type

A service is not required to implement an interface even if you use the interface to access a function in the service.

#### *propertyList*

Every property is optional:

#### **@WSDL**

Is meaningful only when a variable that is based on the Service part is bound to a Web service.

Allows EGL to create the Web Service Description Language (WSDL) definition that is used for interacting with the Java JAX-RPC runtime code. The property fields and types are as follows:

#### **elementName STRING**

If this property field is present, the value sets the following names:

- The WSDL definition file name
- The name in the WSDL portType element
- The name of the binding element; but the element name also has the suffix *Binding*
- The name of the service element; but the element name also has the suffix *Service*

If the property field is not present, the name of the Service part is used for the above-stated purposes.

The data is case sensitive: for example, the name *myService* is different from *MYSERVICE*.

#### **namespace STRING**

Sets the global, target namespace for the WSDL definition. If the property field is not present, the value is created by starting with the string *http://* and inverting every qualifier in the package name; for example, if the package name is *com.ibm.egl*, the value of **namespace** is as follows:

*http://egl.ibm.com*

The value is case sensitive: for example, the namespace *http://egl.ibm.com* is different from *http://EGL.IBM.com*.

#### **isLastParamReturnValue BooleanKind**

This property field is ignored in the context of a Service part.

#### **alias STRING**

Identifies a string that is incorporated into the names of generated output. If you do not set the **alias** property, the service-part name is used instead.

#### **localSQLScope BooleanKind**

Indicates whether identifiers for SQL result sets and prepared statements are local to the service code during invocation by other EGL code, as is the default. If you accept the value *yes*, different programs can use the same identifiers independently, and the EGL code that uses the service can independently use the same identifiers as are used in the service.

If you specify *no*, the identifiers are shared throughout the run unit. The identifiers created when the SQL statements in the service is invoked are available in other code that invokes the service, although the other code can use **localSQLScope = yes** to block access to those identifiers. Also, the

service may reference identifiers created in the invoking code, but only if the SQL-related statements were already run in the other code and if the other code did not block access.

The effects of sharing SQL identifiers are as follows:

- You can open a result set in one code and get rows from that set in another
- You can prepare an SQL statement in one code and run that statement in another

In any case, the identifiers available when the program or pageHandler accesses the service are available when the same program or pageHandler accesses the same or another function in the same service.

*useDeclaration*

Provides easier access to a dataTable or library. For details, see *Use declaration*.

**private**

Indicates that the function is unavailable outside the service. If you omit the term **private**, the function is available.

Variables and constants are necessarily private, but the qualifier is available.

*fieldName*

Name of a primitive field. For the rules of naming, see *Naming conventions*.

*primitiveType*

The primitive type of a field or (in relation to an array) the primitive type of an array element.

*length*

The parameter's length or (in relation to an array) the length of an array element. The length is an integer that represents the number of characters or digits in the memory area referenced either by *fieldName* or (in the case of an array) *dynamicArrayName*.

*decimals*

For a numeric type, you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

*"dateTimeMask"*

For TIMESTAMP and INTERVAL types, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the datetime value. The mask is present with the data at run time.

*dataItemPartName*

The name of a dataItem part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

*recordName*

Name of a record. For the rules of naming, see *Naming conventions*.

*recordPartName*

Name of a record part that is visible to the program. For details on visibility, see *References to parts*.

The part acts as a model of format, as described in *Typedef*.

*constantName literal*

Name and value of a constant. The value is either a quoted string or a number. For the rules of naming, see *Naming conventions*.

*fieldProperty*

A field-specific property-and-value pair, as described in *Overview of EGL properties and overrides*.

*recordProperty*

A record-specific property-and-value pair. For details on the available properties, see the reference topic for the record type of interest.

A basic record has no properties.

*fieldName*

Name of a record field whose properties you wish to override. See *Overview of EGL properties and overrides*.

*arrayName*

Name of a dynamic array. If you use this option, the other symbols to the right (*dataItemPartName*, *primitiveType*, and so on) refer to each element of the array.

*size*

Number of elements in the array. If you specify the number of elements, the array is initialized to the specified number of elements.

*functionPart*

A function. The following restrictions apply:

- The return type cannot be of type ANY, BLOB, or CLOB.
- A parameter cannot include a field of type ANY; cannot be of type ANY, BLOB, CLOB; cannot be of a loose type such as NUMBER; and cannot include the modifier **field** or **nullable**
- Parameters with the modifier OUT must be at the end of the list of parameters

The following function property is valid only in a Service part and is meaningful only when the variable that is based on that part is bound to a Web service:

**@WSDL**

Allows EGL to extract data from the Web Service Description Language (WSDL) definition for use in interacting with the Java JAX-RPC runtime code. The property fields and types are as follows:

**elementName STRING**

If this property field is present, the value becomes the name in the WSDL operation element for the function. If the property field is not present, the function name is used. The data is case sensitive: for example, the name *myFunction* is different from *MYFUNCTION*.

**namespace STRING**

This property field is ignored in the context of a function.

**isLastParamReturnValue BooleanKind**

This property field is ignored in the context of a function.

For details, see *Function part in EGL source format*.

## **Related concepts**

"EGL interfaces" on page 151

“EGL services and Web services” on page 158  
“Library part of type ServiceBindingLibrary” on page 172

#### Related tasks

“Creating an EGL Interface part” on page 150  
“Creating an EGL Service part” on page 157  
“Creating an Interface part from a Service part” on page 154

#### Related reference

“Best practices for services and related interfaces in EGL” on page 162  
“Function part in EGL source format” on page 621  
“Interface part in EGL source format” on page 633  
“Interfaces of type BasicInterface” on page 636

---

## SQL data codes and EGL host variables

The property **SQL data code** identifies the SQL data type to associate with the EGL host variable. The data code is used by the database management system at declaration time, validation time, or generated-program run time.

You may want to vary the SQL data code for a host variable that is of primitive type CHAR, DBCHAR, HEX, or UNICODE. For a host variable of one of the other primitive types, however, SQL data codes are fixed.

If EGL retrieved a column definition from the database management system, do not modify the SQL data code that was retrieved, if any.

The next sections cover these topics:

- “Variable and fixed-length columns”
- “Compatibility of SQL data types and EGL primitive types” on page 875
- “VARCHAR, VARGRAPHIC, and the related LONG data types” on page 876
- “DATE, TIME, and TIMESTAMP” on page 876

### Variable and fixed-length columns

To indicate that a table column is variable length or fixed length, set the SQL data code for the corresponding host variable to the appropriate value, as shown in the next table.

EGL primitive type	SQL data type	Variable or fixed	SQL data code
CHAR	CHAR (the default)	Fixed	453
	VARCHAR, length < 255	Variable	449
	VARCHAR, length > 254	Variable	457
DBCHAR, UNICODE	GRAPHIC (the default)	Fixed	469
	VARGRAPHIC, length < 128	Variable	465
	VARGRAPHIC, length > 127	Variable	473

**Note:** A SQL data type may require the use of null indicators, but this requirement has no effect on how you code an EGL program. For details on nulls, see *SQL support*.

## Compatibility of SQL data types and EGL primitive types

An EGL host variable and the corresponding SQL table column are compatible in any of the following situations:

- The SQL column is any form of character data, and the EGL host variable is of type CHAR with a length less than or equal to the length of the SQL column.
- The SQL column is any form of DBCHAR data, and the EGL host variable is of type DBCHAR with a length less than or equal to the length of the SQL column.
- The SQL column is any form of number and the EGL host variable is of either of these types:
  - BIN, with 2 or 4 bytes and no decimal places.
  - DECIMAL, with a maximum length of 18 digits, including decimal places. The number of digits for a DECIMAL variable should be the same for the EGL host variable and for the column.
  - SMALLINT.
- The SQL column is of any data type, the EGL host variable is of type HEX, and the column and host variable contain the same number of bytes. No data conversion occurs during data transfer.

EGL host variables of type HEX support access to any SQL column of a data type that does not correspond to an EGL primitive type.

If character data is read from an SQL table column into a shorter host variable, content is truncated on the right. To test for truncation, use the reserved word **trunc** in an EGL **if** statement.

If numeric data is read from an SQL table column into a shorter host variable, leading zeros are truncated on the left. If the number still does not fit into the host variable, fractional parts of the number (in decimal) are deleted on the right, with no indication of error. If the number still does not fit, a negative SQL code is returned to indicate an overflow condition.

The next table shows the EGL host variable characteristics that are assigned when the retrieve feature of the EGL editor extracts information from a database management system.

SQL data type	EGL host variable characteristics			SQL data code (SQLTYPE)
	Primitive type	Length	Number of bytes	
BIGINT	HEX	16	8	493
CHAR	CHAR	1–32767	1–32767	453
DATE	CHAR	10	10	453
DECIMAL	DECIMAL	1-18	1–10	485
DOUBLE	HEX	16	8	481
FLOAT	HEX	16	8	481
GRAPHIC	DBCHAR	1–16383	2–32766	469
INTEGER	BIN	9	4	497
LONG VARBINARY	HEX	65534	32767	481
LONG VARCHAR	CHAR	>4000	>4000	457
LONG VARGRAPHIC	DBCHAR	>2000	>4000	473

SQL data type	EGL host variable characteristics			SQL data code (SQLTYPE)
	Primitive type	Length	Number of bytes	
NUMERIC	DECIMAL	1-18	1-10	485
REAL	HEX	8	4	481
SMALLINT	BIN	4	2	501
TIME	CHAR	8	8	453
TIMESTAMP	CHAR	26	26	453
VARBINARY	HEX	2-65534	1-32767	481
VARCHAR	CHAR	≤4000	≤4000	449
VARGRAPHIC	DBCHAR	≤2000	≤4000	465

## VARCHAR, VARGRAPHIC, and the related LONG data types

The definition of an SQL table column of type VARCHAR or VARGRAPHIC includes a maximum length, and the retrieve command uses that maximum to assign a length to the EGL host variable. The definition of an SQL table column of type LONG VARCHAR or VARGRAPHIC, however, does not include a maximum length, and the retrieve command uses the SQL-data-type maximum to assign a length.

## DATE, TIME, and TIMESTAMP

Make sure that the format used for the EGL system default long Gregorian format is the same as the date format specified for the SQL database manager. For details on how the EGL format is set, see *VGVar.currentFormattedGregorianDate*.

You want the two formats to match so that the dates provided by the system variable *VGVar.currentFormattedGregorianDate* are in the format expected by the SQL database manager.

### Related concepts

“SQL support” on page 277

### Related reference

“SQL item properties” on page 68

“currentFormattedGregorianDate” on page 1078

---

## SQL record internals

You need to be aware of the internal layout of an SQL record in any of these situations:

- You use an EGL assignment statement to copy an SQL record to or from a record of a different type
- The runtime argument passed to an EGL program is an SQL record, but the program parameter is not an SQL record
- The runtime argument passed to an EGL function is an SQL record; in this case, the parameter must be a working storage record
- You receive an SQL record as a parameter in a non-EGL program
- You use an SQL record as the original or redefining record in a record redefinition.

Four bytes precede each structure item in an SQL record. The first two bytes are a null indicator, and a null is interpreted as any negative value. The second two bytes are reserved for use as a length field, and you should *not* access that field.

#### Related concepts

“Function part” on page 150

“Program part” on page 148

“SQL support” on page 277

#### Related tasks

“Declaring a record that redefines another” on page 54

#### Related reference

“Assignments” on page 456

---

## SQL record part in EGL source format

You declare a record part of type `sqlRecord` in an EGL source file, which is described in *EGL source format*. For an overview of how EGL interacts with relational databases, see *SQL support*.

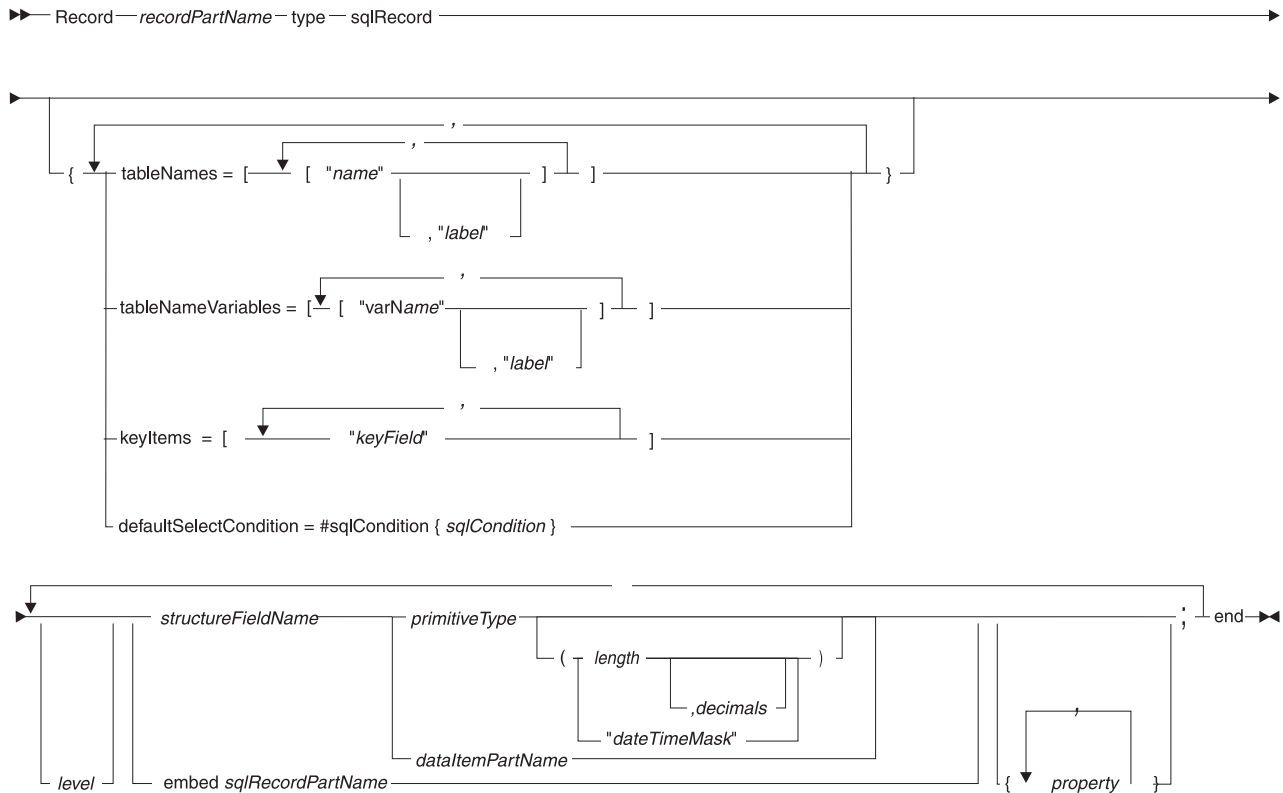
An example of a SQL record part is as follows:

```
Record mySQLRecordPart type sqlRecord
{
  tableNames = ["mySQLTable", "T1"],
  keyItems = ["myHostVar01"],
  defaultSelectCondition =
    #sqlCondition{ // no space between #sqlCondition and the brace
      myHostVar02 = 4 -- start each SQL comment
                    -- with a double hyphen
    }
}

// The structure of an SQL record has no hierarchy
10 myHostVar01 myDataItemPart01
{
  column = "column01",
  isNullable = no,
  isReadOnly = no
};
10 myHostVar02 myDataItemPart02
{
  column = "column02",
  isNullable = yes,
  isReadOnly = no
};
end
```

The syntax diagram for an SQL record part is as follows:





### Record *recordPartName* **sqlRecord**

Identifies the part as a record part of type `sqlRecord` and specifies the name.  
For rules, see *naming conventions*.

#### **tableNames = [ ["name", "label"], ..., ["name", "label"] ]**

Lists the table or tables that are accessed by the SQL record. If you specify a label for a given table name, the label is included in the default SQL statements that are associated with the record.

You may include a double quote mark (") in a table name by preceding the quote mark with the escape character (\). That convention is necessary, for example, when a table name is one of these SQL reserved words:

- CALL
- FROM
- GROUP
- HAVING
- INSERT
- ORDER
- SELECT
- SET
- UPDATE
- UNION
- VALUES
- WHERE

Each of those names must be embedded in a doubled pair of quote marks. If the only table name is *SELECT*, for example, the *tableNames* clause is as follows:

```
tableNames=[["\"SELECT\""]]
```

A similar situation applies when one of those SQL reserved words is used as a column name.

**tableNameVariables** = [{"varName", "label"}, ..., {"varName", "label"}]

Lists one or more table-name variables, each of which contains the name of a table that is accessed by the SQL record. The name of a table is determined only at run time.

The variable may be qualified by a library name and may be subscripted.

If you specify a label for a given table-name variable, the label is included in the default SQL statements that are associated with the record.

You may use table-name variables alone or with table names; but the use of any table-name variable ensures that the characteristics of your SQL statement will be determined only at run time.

You may include a double quote mark (") in a table-name variable by preceding the quote mark with the escape character (\).

**keyItems** = ["item", ..., "item"]

Indicates that the column associated with a given record item is part of the key in the database table. If the database table has a composite key, the order of the record items that are defined as keys must match the order of the columns that are keys in the database table.

**defaultSelectCondition** = #sqlCondition { sqlCondition }

Defines part of the search criterion in the WHERE clause of an implicit SQL statement. The value of *defaultSelectCondition* does not include the SQL keyword WHERE.

EGL provides an implicit SQL statement with a WHERE clause when you code one of these EGL statements:

- **get**
- **open**
- **execute** (only when you request an implicit SQL DELETE or UPDATE statement)

The implicit SQL statements are not stored in the EGL source code. For an overview of those statements, see *SQL support*.

*level*

Integer that indicates the hierarchical position of a structure field. If you exclude this value, the part is a record part; if you include this value, the part is a fixed-record part.

*structureFieldName*

Name of a structure field. For rules, see *Naming conventions*.

*primitiveType*

The primitive type assigned to the structure field.

*length*

The structure field's length, which is an integer. The value of a memory area that is based on the structure item includes the specified number of characters or digits.

### *decimals*

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

### *"dateTimeMask"*

For items of type INTERVAL or TIMESTAMP, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the item value. The mask is present with the data at run time.

### *dataItemPartName*

Specifies the name of a dataItem part that acts as a model of format for the structure item being declared. For details, see *typeDef*.

### **embed** *sqlRecordPartName*

Specifies the name of a record part of type sqlRecord and embeds the structure of that record part into the current record. The embedded structure does not add a level of hierarchy to the current record. For details, see *typeDef*.

### *property*

An item property, as described in *Overview of EGL properties and overrides*. In an SQL record, the SQL field properties are particularly important.

### **Related concepts**

- "EGL projects, packages, and files" on page 15
- "Overview of EGL properties" on page 64
- "Parts" on page 19
- "References to parts" on page 23
- "Record parts" on page 135
- "SQL support" on page 277
- "Typedef" on page 28

### **Related tasks**

- "Syntax diagram for EGL statements and commands" on page 884

### **Related reference**

- "Arrays" on page 75
- "DataItem part in EGL source format" on page 566
- "EGL source format" on page 586
- "Function part in EGL source format" on page 621
- "Indexed record part in EGL source format" on page 632
- "MQ record part in EGL source format" on page 769
- "Naming conventions" on page 778
- "Primitive types" on page 34
- "Program part in EGL source format" on page 841
- "References to variables in EGL" on page 59
- "Relative record part in EGL source format" on page 865
- "Serial record part in EGL source format" on page 868
- "SQL item properties" on page 68

---

## Structure field in EGL source format

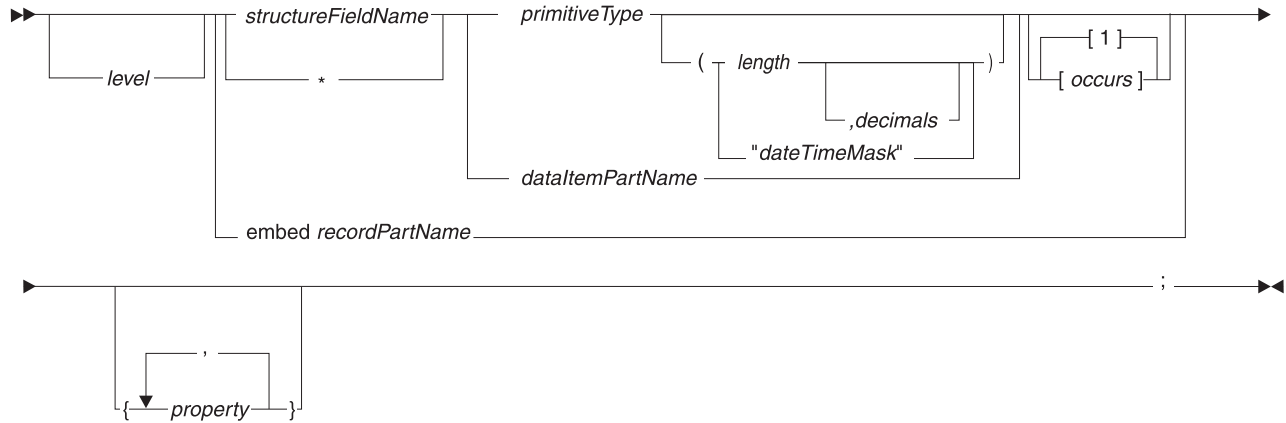
An example of a structure field is as follows:

```

10 address;
20 street01 CHAR(20);
20 street02 CHAR(20);

```

The syntax diagram for a structure field is as follows:



#### *level*

Integer that indicates the hierarchical position of a structure field.

#### *structureFieldName*

Name of a structure field. For rules, see *Naming conventions*.

- \* Indicates that the structure field describes a *filler*, which is a memory area whose name is of no importance. An asterisk is not valid in a reference to an area of memory, as noted in *References to variables and constants*.

#### *primitiveType*

The primitive type assigned to the structure field.

#### *length*

The structure field's length, which is an integer. The value of a memory area that is based on the structure field includes the specified number of characters or digits.

#### *decimals*

For a numeric type (BIN, DECIMAL, NUM, NUMC, or PACF), you may specify *decimals*, which is an integer that represents the number of places after the decimal point. The maximum number of decimal positions is the smaller of two numbers: 18 or the number of digits declared as *length*. The decimal point is not stored with the data.

#### *"dateTimeMask"*

For items of type INTERVAL or TIMESTAMP, you may specify *"dateTimeMask"*, which assigns a meaning (such as "year digit") to a given position in the field value. The mask is present with the data at run time.

#### *dataItemPartName*

Specifies the name of a dataItem part that acts as a model of format for the structure field being declared. For details, see *typeDef*.

#### **embed** *recordPartName*

Specifies the name of a record part and embeds the structure of that record part into the current record. The embedded structure does not add a level of hierarchy to the current record. For details, see *typeDef*.

*recordPartName*

Specifies the name of a record part and includes the structure of that record part in the current record. In the absence of the word *embed*, the record structure is included as a substructure of the structure field being declared. For details, see *typeDef*.

*occurs*

The number of elements in an array of structure items. The default is 1, which means that the structure field is not an array unless you specify otherwise. For details, see Arrays.

*property*

An field property, as described in *Overview of EGL properties and overrides*.

### Related concepts

"Syntax diagram for EGL functions" on page 884

"Overview of EGL properties" on page 64

### Related reference

"Arrays" on page 75

"Naming conventions" on page 778

"Primitive types" on page 34

"References to variables in EGL" on page 59

"Typedef" on page 28

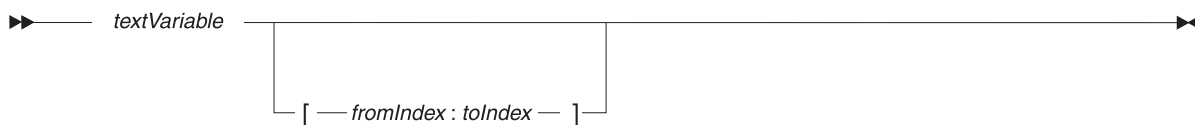
---

## Substrings

In any context in which you reference a character field, you can reference a substring, which is a sequential subset of the characters in that field. If a field value is *ABCD*, you can reference (for example) *BC*, which is the second and third character.

In addition, you can specify a substring on the left side of an assignment statement if the target field is of type CHAR, DBCHAR, or UNICODE. The substring area is filled (padded with blanks, if necessary), and the assigned text does not extend beyond the substring area (but is truncated, if necessary). In addition, you can specify a substring on the left side if the target field is a limited-length string; that situation is described later, by example.

The syntax of a substring reference is as follows.



*itemReference*

An character or HEX field, but not a literal. The item may be a system variable or an array element. Special considerations apply to limited-length strings, as described later.

*fromIndex*

The first character of interest in the item, where 1 represents the first character in the character item, 2 represents the second, and so on. You can use a numeric expression that resolves to an integer, but the expression cannot include a function invocation.

The value of *fromIndex* represents a byte position unless *itemReference* refers to an item of type DBCHAR or UNICODE, in which case the value represents a double-byte character position.

When *itemReference* is a string (not limited length), the value of *fromIndex* is between 1 and the length of the string.

When *itemReference* is a limited-length string, the value of *fromIndex* is between 1 and the length you specified in the variable declaration. If the position has no value, the substring is an empty string, as shown in a later example.

Count from the leftmost character, even if you are working with a bidirectional language such as Arabic or Hebrew.

#### *toIndex*

The last character of interest in the item, where 1 represents the first character in the character item, 2 represents the second, and so on. You can use a numeric expression that resolves to an integer, but the expression cannot include a function invocation.

The value of *toIndex* represents a byte position unless *itemReference* refers to an item of type DBCHAR or UNICODE, in which case the value represents a double-byte character position.

When *itemReference* is a string (not limited length), the maximum value of *toIndex* is the position of the last character in the string.

When *itemReference* is a limited-length string, the maximum value of *toIndex* is the length specified in the variable declaration. If the position has no value, the substring extends only to the last position that holds a character, as shown in a later example.

If *toIndex* is greater than *fromIndex* and both numbers are valid, the substring is an empty string.

Count from the leftmost character, even if you are working with a bidirectional language such as Arabic or Hebrew.

Consider the following example:

```
limited string(20);
s string;
limited = "123456789";
s = limited[11:12]; // No error, and value of s is "" (an empty string).
s = limited[8:12]; // No error, and value of s is "89".
limited = s[8:12];  // Error because s has no length limit.
                  // The last valid position is the one with the last character.
```

If you substring a limited-length string on the left side of an assignment statement and if *fromIndex* is beyond the position that contains the last character, EGL substitutes a blank for each null character that is between the value already in the limited-length string and the assigned content:

```
limited string(20) = "123456789";
s string = "abc";
limited[12:14] = s; // no error; value of limited becomes "123456789  abc"
```

:

#### **Related concepts**

“References to variables in EGL” on page 59

### Related tasks

"Syntax diagram for EGL statements and commands"

### Related reference

"Text expressions" on page 601

---

## Syntax diagram for EGL functions

In the topic that describes a given EGL system function, a syntax diagram gives you details on the type of each function parameter and on the type of value returned, if any. The name of the function library is specified early in the topic.

An example diagram is as follows:

```
StrLib.clip(text STRING in)  
returns (result STRING)
```

The diagram starts with the name of the function and shows a list of parameter specifications, each of which includes the following details:

- The parameter name, which you are free to specify; in this example, the name of the one parameter is *text*.
- The parameter type, which is a type in the EGL language or is a combination of types. (If the type is not in the EGL language, a further description is provided in the topic). In this example, the type is **STRING**.
- The modifier **in**, **out**, or **inOut**, as described in *Function parameters*.

If the parameter specification is surrounded by brackets ([ ]), the argument associated with that parameter is optional. If the specification is surrounded by braces ({}), the argument is also optional, but in this case you can include multiple arguments that are all of the same type.

If the function returns a value, the diagram shows the word *Returns* and a parenthesized name and type. The topic refers to that name when describing the return value, but the name is otherwise meaningless.

If a returns clause is surrounded by brackets ([ ]), the return value is optional.

### Related reference

"Function invocations" on page 613

"Function parameters" on page 616

"EGL library ConsoleLib" on page 886

"EGL library J2EELib" on page 931

"EGL library JavaLib" on page 934

"EGL library LobLib" on page 958

"EGL library MathLib" on page 966

"EGL library ReportLib" on page 990

"EGL library StrLib" on page 996

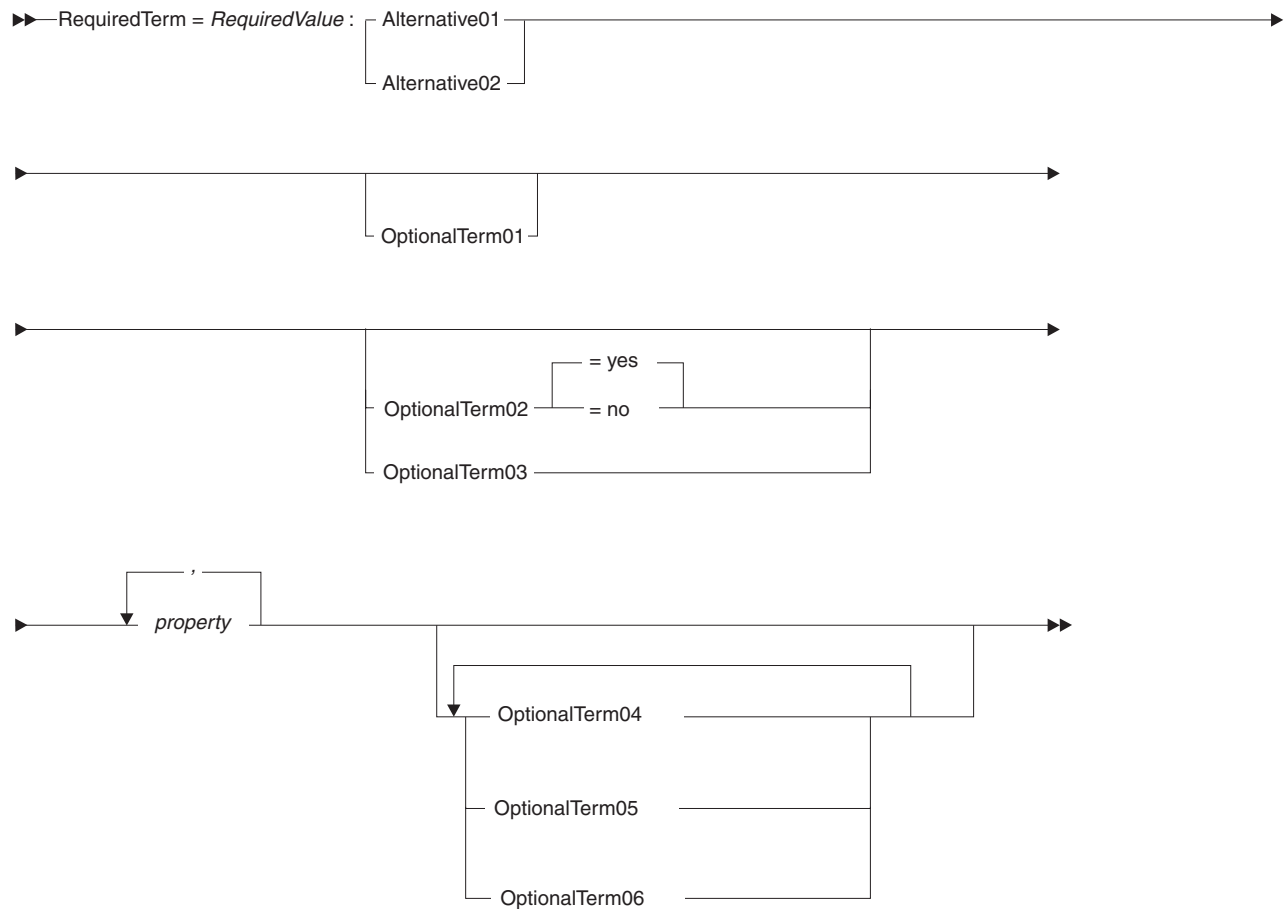
"EGL library SysLib" on page 1016

"EGL library VGLib" on page 1047

---

## Syntax diagram for EGL statements and commands

The IBM syntax diagram lets you see quickly how to construct an EGL statement or build command. An example of such a diagram is as follows:



Read the diagram from left-to-right, top-to-bottom, following the *main path*, which is the line that begins on the left with double arrowheads (>>). As you follow the main path you may select an entry on a subordinate path, in which case you continue reading from left-to-right along the subordinate path.

In the example, the main path is composed of four line segments. It is important to see this. The second and third line segments of the main path each begins with a single arrowhead (>) and includes subordinate information. The fourth line segment of the main path line also begins with a single arrowhead (>), includes returning arrows and subordinate information, and ends with two arrowheads facing each other (><).

A term (or symbol) that is not in italics must be specified exactly as shown. In the example, you specify the term **RequiredTerm** as is. In contrast, a term in italics is a placeholder for a value that you specify. In the example, you might include any of the following symbols in place of *RequiredValue*:

```

myVariable
50
"0h!"

```

The specific requirements for an italicized term (for example, whether a string or number is appropriate) are explained in the text that follows the syntax diagram, not in the syntax diagram itself.



If a diagram shows a non-alphanumeric character, you type that character as part of the syntax. After you specify a value for *RequiredValue*, for instance, you type a colon (:) and a blank.

If you are allowed to select from any of several terms, the terms are shown in a stack. In the example, you can specify the term **Alternative01** or **Alternative02**.

If (as in this case) you *must select* a term from those listed in a stack, one of the choices (arbitrarily specified) is on the top line of the stack. If you are not required to select a term, the terms are all below the top line of the stack, as is true of **OptionalTerm01**.

A value that is on a path but is shown in an elevated way (as is true of = **yes**) is the default value for the stack in which the value appears. The example indicates that you can specify any of the following strings, and the first two are equivalent:

```
optionalTerm01 = yes
```

```
optionalTerm01
```

```
optionalTerm01 = no
```

```
OptionalTerm02
```

An arrow returning to the left above a term indicates that you can use the term repeatedly. In the example, you specify values for *property*, each separated from the next with a comma.

An arrow returning to the left above a vertical stack means that you can choose from the list of entries in any order. In the example, each of the following strings is valid (as are other variations), but none is required:

```
OptionalTerm04 OptionalTerm05
```

```
OptionalTerm06
```

```
OptionalTerm04 OptionalTerm06 OptionalTerm05
```

---

## System Libraries

### EGL library ConsoleLib

The Console library provides the consoleUI functionality to EGL programs. Using the **ConsoleLib** qualifier (for example, **ConsoleLib.activateWindow**) is optional.

Function	Description
activateWindow ( <i>window</i> )	Makes the specified window the active window, and updates the <b>ConsoleLib</b> variable <i>activeWindow</i> accordingly.
activateWindowByName ( <i>name</i> )	Makes the specified window the active window, and updates the <b>ConsoleLib</b> variable <i>activeWindow</i> accordingly.
cancelArrayDelete ()	Terminates the current <i>delete</i> operation in progress during the execution of a <b>BEFORE_DELETE OpenUI</b> event code block.

Function	Description
<code>cancelArrayInsert ()</code>	Terminates the current <i>insert</i> operation in progress during the execution of a <b>BEFORE_INSERT OpenUI</b> event code block.
<code>clearActiveForm ()</code>	Clears the display buffers of the all of the fields.
<code>clearActiveWindow ()</code>	Removes all displayed material from the active window.
<code>clearFields ([<i>consoleField</i>{, <i>consoleField</i>})</code>	Clears the display buffers of the specified fields in the active form. If no fields are specified, all fields of the form are cleared.
<code>clearFieldsByName (<i>fieldName</i>{, <i>fieldName</i>})</code>	Clears the display buffers of the named fields in the active form. If no fields are named, all fields of the form are cleared.
<code>clearForm (<i>consoleForm</i>)</code>	Clears the display buffers of the all of the fields.
<code>clearWindow (<i>window</i>)</code>	Removes all displayed material from the specified window.
<code>clearWindowByName (<i>name</i>)</code>	Removes all displayed material from the specified window.
<code>closeActiveWindow ()</code>	Clears the window from the screen, releases the resources associated with that window, and activates the previous active window.
<code>closeWindow (<i>window</i>)</code>	Clears the window from the screen, releases the resources associated with that window, and activates the previous active window.
<code>closeWindowByName (<i>name</i>)</code>	Clears the window from the screen, releases the resources associated with that window, and activates the previous active window
<code>result = currentArrayCount ()</code>	Returns the number of elements in the dynamic array that is associated with the current active form
<code>result = currentArrayDataLine ()</code>	Returns the number of the program record within the program array that is displayed in the current line of a screen array during or immediately after the <b>OpenUI</b> statement.
<code>result = currentArrayScreenLine ()</code>	Returns the number of the current screen record in its screen array during an <b>OpenUI</b> statement.
<code>displayAtLine (<i>text</i>, <i>line</i>)</code>	Displays a string to a specified place within the active window.
<code>displayAtPosition (<i>text</i>, <i>line</i>, <i>column</i>)</code>	Displays a string to a specified place within the active window.
<code>displayError (<i>msg</i>)</code>	Causes the error window to be created and shown and displays the error message in that window.
<code>displayFields ([<i>consoleField</i>{, <i>consoleField</i>})</code>	Displays form field values to the Console.

Function	Description
<code>displayFieldsByName (consoleFieldName{, consoleFieldName})</code>	Displays form field values to the Console.
<code>displayForm (consoleForm)</code>	Displays the form to the active window.
<code>displayFormByName (formName)</code>	Displays the form to the active window.
<code>displayLineMode (text)</code>	Displays a string in <i>line mode</i> rather than <i>form/window mode</i> .
<code>displayMessage (msg)</code>	Displays a string to a specified place within the active window and uses the <i>messageLine</i> settings of the active window to identify where to display the string.
<code>drawBox (row, column, depth, width)</code>	Draws a rectangle in the active window with the specified location and dimensions.
<code>drawBoxWithColor (row, column, depth, width, Color)</code>	Draws a rectangle in the active window with the specified location, dimensions, and color.
<code>result = getKey ()</code>	Reads a key from the input and returns the integer code for the key.
<code>result = getKeyCode (keyName)</code>	Returns the key integer code of the named key in the String.
<code>result = getKeyName (keyCode)</code>	Returns the name that represents the integer key code.
<code>gotoField (consoleField)</code>	Moves the cursor to the specified form field.
<code>gotoFieldByName (name)</code>	Moves the cursor to the specified form field.
<code>gotoMenuItem (item)</code>	Moves the menu cursor to the specified menu item.
<code>gotoMenuItemByName (name)</code>	Moves the menu cursor to the specified menu item.
<code>hideAllMenuItems ()</code>	Hides all menu items in the currently displayed menu.
<code>hideErrorWindow ()</code>	Hides the error window.
<code>hideMenuItem (item)</code>	Hides a specified menu item so that a user cannot select it.
<code>hideMenuItemByName (name)</code>	Hides a specified menu item so that a user cannot select it.
<code>result = isCurrentField (consoleField)</code>	Returns <b>true</b> if the cursor is in the specified form field; otherwise it returns <b>false</b> .
<code>result = isCurrentFieldByName (name)</code>	Returns <b>true</b> if the cursor is in the specified form field; otherwise it returns <b>false</b> .
<code>result = isFieldModified (consoleField)</code>	Returns <b>true</b> if the user changed the contents of the specified form field; a <b>false</b> return indicates that the field has not been edited.

Function	Description
<i>result</i> = <code>isFieldModifiedByName (name)</code>	Returns <b>true</b> if the user changed the contents of the specified form field; a <b>false</b> return indicates that the field has not been edited.
<i>result</i> = <code>lastKeyTyped ()</code>	Returns the integer code of the last physical key that was pressed on the keyboard.
<code>nextField ()</code>	Moves the cursor to the next form field according to the defined field travel order.
<code>openWindow (window)</code>	Makes a window visible and adds it to the top of the window stack. The form is displayed in the window.
<code>openWindowByName (name)</code>	Makes a window visible and adds it to the top of the window stack.
<code>openWindowWithForm (window, form)</code>	Makes a window visible and adds it to the top of the window stack. The Window size will change to hold the specified form if the window size was not defined when the window was declared.
<code>openWindowWithFormByName (windowName, formName)</code>	Makes a window visible and adds it to the top of the window stack.
<code>previousField ()</code>	Moves the cursor to the previous form field according to the defined field travel order.
<i>result</i> = <code>promptLineMode (prompt)</code>	Displays a prompt message to the user in a <i>line mode</i> environment.
<code>scrollDownLines (numLines)</code>	Scrolls the on-screen data toward the bottom of the data by the specified number of lines.
<code>scrollDownPage ()</code>	Scrolls the on-screen data one page toward the bottom of the data.
<code>scrollUpLines (numLines)</code>	Scrolls the on-screen data toward the top of the data by the specified number of lines.
<code>scrollUpPage ()</code>	Scrolls the on-screen data one page toward the top of the data.
<code>setArrayLine (recordNumber)</code>	Moves the selection to the specified program record. The data table is scrolled in the display if necessary to make the selected record visible.
<code>setCurrentArrayCount (count )</code>	Sets how many records exist in the program array. Must be called prior to the <b>OpenUI</b> statement.
<code>showAllMenuItems ()</code>	Shows the all menu items for user selection.
<code>showHelp (helpkey)</code>	Displays the <b>ConsoleUI</b> help screen during execution of the EGL program.
<code>showMenuItem (item)</code>	Shows the specified menu item for user selection.
<code>showMenuItemByName(name)</code>	Shows the specified menu item for user selection.

Function	Description
updateWindowAttributes( <i>attribute</i> {, <i>attribute</i> })	Updates the current active window one or more attributes.

Variables	Description
activeForm	The most recently displayed form in the active window.
activeWindow	The topmost window, and it is the target for window operations when no window name is specified.
commentLine	The window line where comment messages are displayed.
CurrentDisplayAttrs	Settings applied to elements displayed through the display functions.
currentRowAttrs	Highlight attributes applied to the current row.
cursorWrap	If <b>true</b> , the cursor wraps around to the first field on the form; if <b>false</b> , the statement ends when the cursor moves forwards from the last input field on the form.
defaultDisplayAttributes	Default settings of <i>presentation attributes</i> for new objects.
defaultInputAttributes	The default settings of <i>presentation attributes</i> for input operations.
deferInterrupt	If <b>true</b> , the program catches <b>INTR</b> signals and logs them in the <i>interruptRequested</i> variable, which the program is then responsible to monitor. On Windows, the signal is simulated when the logical <b>INTERUPT</b> key is pressed, which is <b>CONTROL_C</b> by default.
deferQuit	If <b>true</b> , the program catches <b>QUIT</b> signals and logs them in the <i>interruptRequested</i> variable, which the program is then responsible to monitor. On Windows, the signal is simulated when the logical <b>QUIT</b> key is pressed, which is <b>CONTROL_</b> by default.
definedFieldOrder	If <b>true</b> , the up and down arrow keys move to the previous and next fields in the traversal order. If <b>false</b> , up and down move to the field in that direction physically on the screen.
errorLine	The window where error messages are displayed.
errorWindow	The window location where error messages are displayed in the ConsoleUI screen.
errorWindowVisible	If <b>true</b> , the error window is currently being displayed to the screen

Variables	Description
formLine	The window line where forms are displayed.
interruptRequested	This indicates that an <b>INTR</b> signal has been received (or simulated).
key_accept	Key for successful termination of <b>OpenUI</b> statements. Default key is <b>ESCAPE</b> .
key_deleteLine	Key for deleting the current row from a screen array. Default key is <b>F2</b> .
key_help	Key for showing context sensitive help during <b>OpenUI</b> statements. default key is <b>CTRL_W</b> .
key_insertLine	Key for inserting a row into a screen array. Default key is <b>F1</b> .
key_interrupt	Key for simulating an <b>INTR</b> signal. Default key is <b>CTRL_C</b> .
key_pageDown	Key for paging forwards in a screen array (data table). Default key is <b>F3</b> .
key_pageUp	Key for paging backwards in a screen array (data table). Default key is <b>F4</b> .
key_quit	Key for simulating a <b>QUIT</b> signal. Default key is <b>CTRL_\\</b> .
menuLine	The window line where menus are displayed.
messageLine	The window line where messages are displayed.
messageResource	The file name of the resource bundle.
promptLine	The number of the line at which prompts are displayed in a window
quitRequested	Indicates that a <b>QUIT</b> signal has been received (or simulated).
screen	Automatically-defined, default, borderless window; the dimensions are equal to the dimensions of the available display surface.
sqlInterrupt	If <b>yes</b> , the user can interrupt SQL statements being processed. If <b>no</b> , the user can only interrupt <b>OpenUI</b> statements. Used in combination with the <i>deferInterrupt</i> and <i>deferQuit</i> variables.

## activeForm

The system variable **ConsoleLib.activeForm** is the most recently displayed form in the active window.

Type: ConsoleForm

### Related reference

“EGL library ConsoleLib” on page 886

## activateWindow()

The system function **ConsoleLib.activateWindow** makes the specified window the active window, and updates the variable *activeWindow*.

**ConsoleLib.activateWindow**(*window1* **Window** inOut)

*window1*

The window to activate.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## activeWindow

The system variable **ConsoleLib.activeWindow** is the topmost window or the one most recently activated. **ConsoleLib.activeWindow** is the target for window operations when no window name is specified.

Type: Window

### Related reference

“EGL library ConsoleLib” on page 886

## activateWindowByName()

The system function **ConsoleLib.activateWindowByName** makes the specified window the active window, and updates the **consoleLib** variable *activeWindow* accordingly.

**ConsoleLib.activateWindowByName**(*name* **STRING** in)

*name*

The name of the window.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## cancelArrayDelete()

The system function **ConsoleLib.cancelArrayDelete** terminates the current *delete* operation in progress during the execution of a **BEFORE\_DELETE OpenUI** event code block.

If at runtime, this function is executed outside the scope of an **OpenUI** statement, the effect is a null operation.

**ConsoleLib.cancelArrayDelete**( )

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## cancelArrayInsert()

The system function **ConsoleLib.cancelArrayInsert** terminates the current *insert* operation in progress during the execution of a **BEFORE\_INSERT OpenUI** event code block. If at runtime, this function is executed outside the scope of an **OpenUI** statement, the effect is a null operation.

```
ConsoleLib.cancelArrayInsert( )
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## clearActiveForm()

The system function **ConsoleLib.clearActiveForm** clears the display buffers of all fields. This function has no effect on the bound data elements; data stored in the bound data elements is not cleared.

```
ConsoleLib.clearActiveForm( )
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## clearActiveWindow

The system function **ConsoleLib.clearActiveWindow** removes all displayed material from the active window. This includes erasing the constant information displayed in the current form. If the active window has a border, the border is not erased. The statement does not affect the ordering of the window stack or affect any windows that are above it in the window stack.

```
ConsoleLib.clearActiveWindow( )
```

### Related reference

“EGL library ConsoleLib” on page 886

## clearFields()

The system function **ConsoleLib.clearFields** clears the display buffers of the specified fields. If no fields are specified, all fields are cleared. This function has no effect on the bound data elements; any data that was stored in the bound data elements will not be cleared.

```
ConsoleLib.clearFields(  
    [consoleField1 ConsoleField inOut  
    {, consoleField1 ConsoleField inOut}  
    ] )
```

*consoleField1*

The name of the variable of type ConsoleField.

### Related concepts

“Syntax diagram for EGL functions” on page 884



#### Related reference

“EGL library ConsoleLib” on page 886

### clearFieldsByName()

The system function **ConsoleLib.clearFieldsByName** clears the specified on-screen fields; and clears all fields if no fields are specified. The variables bound to the on-screen fields are not affected.

```
ConsoleLib.clearFieldsByName(  
    [fieldName STRING in  
    { , fieldName STRING in}])
```

*fieldName*

The value of a ConsoleField name field.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### clearForm()

The system function **ConsoleLib.clearForm** clears all fields in the specified form. The variables bound to those fields are not affected.

```
ConsoleLib.clearForm(consoleForm ConsoleForm inOut)
```

*consoleForm*

A variable of type ConsoleForm.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### clearWindow()

The system function **ConsoleLib.clearWindow** removes all displayed material from the specified window. This includes erasing the constant information displayed in the current form. If the window has a border, the border is not erased. The statement does not affect the ordering of the window stack or affect any windows that are above it in the window stack.

```
ConsoleLib.clearWindow(window1 Window inOut)
```

*window1*

The window to be cleared.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### clearWindowByName()

The system function **ConsoleLib.clearWindowByName** removes all displayed material from the specified window. This includes erasing the constant information

displayed in the current form. If the window has a border, the border is not erased. The statement does not affect the ordering of the window stack. The **ActiveWindow** variable refers to the topmost window in the display stack.

**ConsoleLib.cleaWindowByName**(*name* **STRING** in)

*name*

The name of the window.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **closeActiveWindow()**

The system function **ConsoleLib.closeActiveWindow** clears the window from the screen, releases the resources associated with the window that was cleared, and activates the previously-active window.

After **ConsoleLib.closeActiveWindow** is invoked, the window cannot be reopened by **ConsoleLib.openWindow** or **ConsoleLib.openWindowByName**. In addition, closing the SCREEN window is not allowed.

**ConsoleLib.closeActiveWindow**( )

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **closeWindow()**

The Console library function **ConsoleLib.closeWindow** clears the specified window from the screen, releases the resources associated with the cleared window, and activates the previously-active window.

After **ConsoleLib.closeWindow** is invoked, the window cannot be reopened by **ConsoleLib.openWindow** or **ConsoleLib.openWindowByName**. In addition, closing the SCREEN window is not allowed.

**ConsoleLib.closeWindow**(*window1* **Window** inOut)

*window1*

The specified window object on the screen.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **closeWindowByName()**

The system function **ConsoleLib.closeWindowByName** clears the named window from the screen, releases the resources associated with the closed window, and activates the previously active window.

After **ConsoleLib.closeWindowByName** is invoked, the window cannot be reopened by **ConsoleLib.openWindow** or **ConsoleLib.openWindowByName**. The console window remains open.

**ConsoleLib.closeWindowByName**(*name* **STRING** *in*)

*name*

The name of the window.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### commentLine

The system variable **ConsoleLib.commentLine** is the window line where comment messages are displayed.

Type: Integer

#### Related reference

“EGL library ConsoleLib” on page 886

### currentArrayCount()

The system function **ConsoleLib.currentArrayCount** returns the number of elements in the dynamic array that is associated with the current active form.

It is recommended that you avoid using this function, which is used to help migrate applications that were written with Informix 4GL. Instead, use the array-specific function **getSize**, as described in *Arrays*.

**ConsoleLib.currentArrayCount**( )  
returns (*result* **INT**)

*result*

The number of elements.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“Arrays” on page 75

“EGL library ConsoleLib” on page 886

### currentArrayDataLine()

The system function **ConsoleLib.currentArrayDataLine** returns the number of the program record within the program array that is displayed in the current line of a screen array during or immediately after the **openUI** statement.

**ConsoleLib.currentArrayDataLine**( )  
returns (*result* **INT**)

*result*

Any integer.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### currentArrayScreenLine()

The system function **ConsoleLib.currentArrayScreenLine** returns the number of the current screen record in its screen array during an **openUI** statement.

```
ConsoleLib.currentArrayScreenLine( )  
returns (result INT)
```

*result*

Any integer.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### currentDisplayAttrs

The system variable **ConsoleLib.currentDisplayAttrs** specifies display characteristics of any text that will be shown after the variable has been set.

Variables of type **PresentationAttributes** include the fields **color**, **intensity**, and **highlight**. For details, see *ConsoleUI parts and related variables*.

Type: **PresentationAttributes**

#### Related reference

“EGL library ConsoleLib” on page 886

“ConsoleUI parts and related variables” on page 209

### currentRowAttrs

The system variable **ConsoleLib.currentRowAttrs** are highlight attributes applied to the current row of a screen array.

Variables of type **PresentationAttributes** include the fields **color**, **intensity**, and **highlight**. For details, see *ConsoleUI parts and related variables*.

Type: **PresentationAttributes**

#### Related reference

**currentDisplayAttrs**

“EGL library ConsoleLib” on page 886

### cursorWrap

The system variable **ConsoleLib.cursorWrap** indicates whether the cursor wraps around to the first field on the form after the user attempts to navigate beyond the last field. The navigation is attempted when the user presses **Tab** or **Enter** or (when **autonext** is set) when the user fills the field.

Valid values are **yes** (in which case the cursor wraps) and **no** (in which case the user’s action causes acceptance of the form).

Type: Boolean

**Related reference**

“EGL library ConsoleLib” on page 886

## **defaultDisplayAttributes**

The system variable **ConsoleLib.defaultDisplayAttributes** contains the settings used for *PresentationAttributes* in variables.

Variables of type *PresentationAttributes* include the fields color, intensity, and highlight. For details, see *ConsoleUI parts and related variables*.

Type: *PresentationAttributes*

**Related reference**

“EGL library ConsoleLib” on page 886

## **defaultInputAttributes**

The system variable **ConsoleLib.defaultInputAttributes** contains the default settings of presentation attributes for input operations.

Variables of type *PresentationAttributes* include the fields color, intensity, and highlight. For details, see *ConsoleUI parts and related variables*.

Type: *PresentationAttributes*

**Related reference**

“EGL library ConsoleLib” on page 886

## **deferInterrupt**

The Console UI library variable **ConsoleLib.deferInterrupt** identifies the behavior of the application when it receives the **INTERRUPT** signal. If the results are **true**, the program catches **INTR** signals and logs them in the *interruptRequested* variable, which the program is then responsible to monitor. On Windows, the signal is simulated when the logical **INTERRUPT** key is pressed, which is **CONTROL\_C** by default. If the results are **false**, the program ends when the **interrupt** key is pressed.

Type: Boolean

**Related reference**

“EGL library ConsoleLib” on page 886

## **deferQuit**

For the system variable **ConsoleLib.deferQuit**, if **true**, the program catches **QUIT** signals and logs them in the *quitRequested* variable, which the program is then responsible to monitor. On Windows, the signal is simulated when the logical **QUIT** key is pressed, which is **CONTROL\_** by default. If **false**, receiving a quit signal will terminate the application.

Type: Boolean

#### Related reference

“EGL library ConsoleLib” on page 886

### definedFieldOrder

The Console UI variable **ConsoleLib.definedFieldOrder** determines the behavior of the up/down arrow keys when inputting with a form. If **true**, the cursor traverses fields in the order of definition when using the up/down arrow keys. If **false**, the cursor moves up and down according to the physical arrangement of the fields on the screen.

Type: Boolean

#### Related reference

“EGL library ConsoleLib” on page 886

### displayAtLine()

The system function **ConsoleLib.displayAtLine** displays a string to a specified place within the active window.

```
ConsoleLib.displayAtLine(  
    text STRING in,  
    line INT in)
```

*text*

The string to display.

*line*

The number of the line on which to display the string.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### displayAtPosition()

The system function **ConsoleLib.displayAtPosition** displays a string to a specified place within the active window.

```
ConsoleLib.displayAtPosition(  
    text STRING in,  
    line INT in,  
    column INT in)
```

*text*

The string to display.

*line*

The number of the line at which to display the string.

*column*

The number of the column on which to display the string.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

## displayError()

The system function **ConsoleLib.displayError** causes the error window to be created and shown, and display the error message in that window. The error window floats above all other windows until it is closed by calling *hideErrorWindow()* or when a key is pressed. If applicable, the terminal bell will be activated.

```
ConsoleLib.displayError(msg STRING in)
```

*msg*

The error message to display.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## displayFields()

The system function **ConsoleLib.displayFields** displays form field values to the Console. If data elements are bound to the fields, the data will be retrieved from those elements and formatted according to the rules specified with the form field. For an unbound form field, data can be set directly to the fields by accessing the **ConsoleField.value** field.

```
ConsoleLib.displayFields(  
  [consoleField1 ConsoleField in  
  { , consoleField1 ConsoleField in }  
  ])
```

*consoleField1*

The name of the variable of type ConsoleField.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## displayFieldsByName()

The system function **ConsoleLib.displayFieldsByName** displays form field values to the Console. If data elements are bound to the fields, the data will be retrieved from those elements and formatted according to the rules specified with the form field. For an unbound form field, data can be set directly to these fields by accessing the **ConsoleField.value** field.

```
ConsoleLib.displayFieldsByName(  
  consoleFieldName1 ConsoleFieldName in  
  { , consoleFieldName1 ConsoleFieldName in } )
```

*consoleFieldName1*

The names of the fields to display.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## displayForm()

The system function **ConsoleLib.displayForm** displays the specified form to the active window.

**ConsoleLib.displayForm**(*consoleForm* **ConsoleForm** in)

*consoleForm*

The name of the variable of type ConsoleForm.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## displayFormByName()

The system function **ConsoleLib.displayFormByName** displays the named form to the active window.

**ConsoleLib.displayFormByName**(*formName* **STRING** in)

*formName*

The value of the ConsoleForm name field.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## displayLineMode()

The system function **ConsoleLib.displayLineMode** displays the designated string in **line mode** rather than **form/window mode**. The string value is sent to the standard *out* location on the running system. All display characteristics such as wrapping and scrolling become the responsibility of the standard output interface.

**ConsoleLib.displayLineMode**(*text* **STRING** in)

*text*

The string to display

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## displayMessage()

The system function **ConsoleLib.displayMessage** displays a string to the message line of the active window. The function uses the *MessageLine* settings of the active window to know where to display the string.

**ConsoleLib.displayMessage**(*msg* **STRING** in)

*msg*

The message to display.



### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## drawBox()

The system function **ConsoleLib.drawBox** draws a rectangle in the active window with the upper-left corner at *row*, *column* for the first two integers and *depth*, *width* for the next two integers. The row and column are relative to the upper-left corner of the current window.

```
ConsoleLib.drawBox(  
    row INT in,  
    column INT in,  
    depth INT in,  
    width INT in)
```

*row*

The row number relative to the upper left corner of the window.

*column*

The column number relative to the upper left corner of the window.

*depth*

The depth or height of the box.

*width*

The width of the box.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## drawBoxWithColor()

The system function **ConsoleLib.drawBoxWithColor** draws a rectangle in the active window with the upper-left corner at *row*, *column* for the first two integers and *depth*, *width* for the next two integers. The row and column are relative to the upper-left corner of the current window. The rectangle is drawn in the specified color.

```
ConsoleLib.drawBoxWithColor(  
    row INT in,  
    column INT in,  
    depth INT in,  
    width INT in,  
    color enumerationColorKind in)
```

*row*

The row number relative to the upper left corner of the window.

*column*

The column number relative to the upper left corner of the window.

*depth*

The depth or height of the box.

*width*

The width of the box.

*color*

The color of the box.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library ConsoleLib” on page 886

**errorLine**

The Console UI variable **ConsoleLib.errorLine** controls the line location where error messages are displayed in the **ConsoleUI** screen.

Type: INT

**Related reference**

“EGL library ConsoleLib” on page 886

**errorWindow**

The system variable **ConsoleLib.errorWindow** is the window where an error message from **ConsoleLib.displayError()** is shown.

Type: Window

**Related reference**

“EGL library ConsoleLib” on page 886

“displayError()” on page 900

**errorWindowVisible**

The Console UI variable **ConsoleLib.errorWindowVisible** identifies the status of the error message window. If **true**, the window is visible. If **false**, the window is not visible.

Type: Boolean

**Related reference**

“EGL library ConsoleLib” on page 886

**formLine**

The system variable **ConsoleLib.formLine** is the default line location where a form is displayed in window. It affects the properties of windows when they are opened.

Type: INT

**Related reference**

“EGL library ConsoleLib” on page 886

**getKey()**

The system function **ConsoleLib.getKey** waits for a key to be pressed and returns the integer code of the physical key that was pressed. This function reads a key

from the input. Results may be interpreted in a portable way by comparing the result with the value returned by **getKeyCode(String keyname)**.

**ConsoleLib.getKey( )**  
returns (*result* INT)

*result*

An integer that represents the key pressed.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“EGL library ConsoleLib” on page 886

“getKey()” on page 903

### **getKeyCode()**

The system function **ConsoleLib.getKeyCode** returns the key integer code of the specified key name.

**ConsoleLib.getKeyCode(keyName STRING in)**  
returns (*result* INT)

*result*

An integer that represents the key name.

*keyName*

The name of the logical or physical key for which to calculate the corresponding key code.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **getKeyName()**

The system function **ConsoleLib.getKeyName** returns the name of the key that represents the integer key code.

**ConsoleLib.getKeyName(keyCode INT in)**  
returns (*result* STRING)

*result*

The name of the key of the integer key code.

*keyCode*

The key integer code.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **gotoField()**

The system function **ConsoleLib.gotoField** moves the cursor to the specified form field. This function is valid in an **OpenUI** statement that acts on a console form.

**ConsoleLib.gotoField(consoleField1 ConsoleField in)**

*consoleField1*

The name of the variable of type `ConsoleField` to which the cursor moves.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library `ConsoleLib`” on page 886

## **gotoFieldByName()**

The system function **`ConsoleLib.gotoFieldByName`** moves the cursor to the specified form field. This function is valid in an **`openUI`** statement that acts on a console form.

```
ConsoleLib.gotoFieldByName(name STRING in)
```

*name*

The name of the field to which the cursor moves.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library `ConsoleLib`” on page 886

## **gotoMenuItem()**

The system function **`ConsoleLib.gotoMenuItem`** moves the menu cursor to the specified menu item. When the function is invoked, the menu item that is specified is selected.

```
ConsoleLib.gotoMenuItem(item MenuItem in)
```

*item*

The menu item to which the cursor moves.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library `ConsoleLib`” on page 886

## **gotoMenuItemByName()**

The system function **`ConsoleLib.gotoMenuItemByName`** moves the menu cursor to the specified menu item. When the function is invoked, the menu item that is specified is selected.

```
ConsoleLib.gotoMenuItemByName(name STRING in)
```

*name*

The name of the menu item to which the cursor moves.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library `ConsoleLib`” on page 886

## hideAllMenuItems()

The system function **ConsoleLib.hideAllMenuItems** hides all menu items in the currently displayed menu.

```
ConsoleLib.hideAllMenuItems( )
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## hideErrorWindow()

The system function **ConsoleLib.hideErrorWindow** hides the error window.

```
ConsoleLib.hideErrorWindow( )
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## hideMenuItem()

The system function **ConsoleLib.hideMenuItem** hides the specified menu item so that the user cannot select it. By default all menu items are shown. The hidden item will not be activated by keystrokes.

```
ConsoleLib.hideMenuItem(item MenuItem in)
```

*item*

The menu item to be hidden.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## hideMenuItemByName()

The system function **ConsoleLib.hideMenuItemByName** hides the specified menu item so that the user cannot select it. By default all menu items are shown. The hidden item will not be activated by keystrokes.

```
ConsoleLib.hideMenuItemByname(name STRING in)
```

*name*

The name of the menu item to be hidden.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## interruptRequested

The Console UI variable **ConsoleLib.interruptRequested** indicates if an INTR signal has been received or simulated. If **true**, an INTR signal has been received. If **false**, an INTR signal has not been received.

Type: Boolean

### Related reference

“EGL library ConsoleLib” on page 886

## isCurrentField()

The system function **ConsoleLib.isCurrentField** returns **yes** if the cursor is in the field and returns **no** if the cursor is not in the field. This function is valid in an **OpenUI** statement that acts on an arrayDictionary.

**ConsoleLib.isCurrentField**(*consoleField1* **ConsoleField** in)  
returns (*result* **BOOLEAN**)

*result*

**true**, if the cursor is in the specified form field; otherwise **false**.

*consoleField1*

The name of the variable of type ConsoleField.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## isCurrentFieldByName()

The system function **ConsoleLib.isCurrentFieldByName** returns **yes** if the cursor is in the field; otherwise returns **no**.

This function is valid in an **OpenUI** statement that acts on a console form.

**ConsoleLib.isCurrentFieldByName**(*name* **STRING** in)  
returns (*result* **BOOLEAN**)

*result*

**true**, if the cursor is in the specified form field; otherwise **false**.

*name*

The value of the ConsoleField name field.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## isFieldModified()

The system function **ConsoleLib.isFieldModified** identifies for **OpenUI** form/fields, whether a field has been modified during the current **OpenUI** Statement. For **OpenUI** screenarray (arrayDictionary), it returns whether the field in the current row has been modified since the cursor entered the row.

This function is valid on commands that modify fields and does not register the effect of statements that appear in a **BEFORE\_OPENUI** clause. You can assign values to fields in these clauses without marking the fields as touched.

**ConsoleLib.isFieldModified**(*consoleFiled1* **ConsoleField** in)  
returns (*result* **BOOLEAN**)

*result*

**true**, if the the specified form field was modified; otherwise **false**.

*consoleFiled1*

The name of the variable of type ConsoleField.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"EGL library ConsoleLib" on page 886

### isFieldModifiedByName()

The system function **ConsoleLib.isFieldModifiedByName** identifies whether or not the contents of a named field have been modified.

**ConsoleLib.isFieldModifiedByName** returns **yes** if the user changed the contents of a field and returns **no** if the user did not change the field contents.

This function is valid on commands that modify fields and does not register the effect of statements that appear in a **BEFORE\_OPENUI** clause. You can assign values to fields in these clauses without marking the fields as touched.

**ConsoleLib.isFieldModifiedByName**(*name* **STRING** in)  
returns (*result* **BOOLEAN**)

*result*

**true**, if the the specified form field was modified; otherwise **false**.

*name*

The value of the ConsoleField name field.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"EGL library ConsoleLib" on page 886

### key\_accept

The system variable **ConsoleLib.key\_accept** is the key for successful termination of a **OpenUI** statement. The default key is **Esc**.

Type: CHAR(32)

#### Related reference

"EGL library ConsoleLib" on page 886

### key\_deleteLine

The system variable **ConsoleLib.key\_deleteLine** is the key for deleting the current row from an arrayDictionary in a console form. The default key is **F2**.

Type: CHAR(32)

**Related reference**

“EGL library ConsoleLib” on page 886

**key\_help**

The system variable **ConsoleLib.key\_help** is the key for showing context-sensitive help during an **OpenUI** statement. The default key is **CRTL\_W**.

Type: CHAR(32)

**Related reference**

“EGL library ConsoleLib” on page 886

**key\_insertLine**

The system variable **ConsoleLib.key\_insertLine** identifies the keystroke used to insert a row in an arrayDictionary on a consoleForm. The default key is **F1**.

Type: CHAR(32)

**Related reference**

“EGL library ConsoleLib” on page 886

**key\_interrupt**

The system variable **ConsoleLib.key\_interrupt** is the key for simulating an interrupt. The default key is **CTRL\_C**.

Type: CHAR(32)

**Related reference**

“EGL library ConsoleLib” on page 886

**key\_pageDown**

The system variable **ConsoleLib.key\_pageDown** is the key that pages forward in an arrayDictionary on a console form. The default key is **F3**.

Type: CHAR(32)

**Related reference**

“EGL library ConsoleLib” on page 886

**key\_pageUp**

The system variable **ConsoleLib.key\_pageUp** is the key for paging backward in an arrayDictionary on a console form. The default key is **F4**.

Type: CHAR(32)

**Related reference**

“EGL library ConsoleLib” on page 886

**key\_quit**

The system variable **ConsoleLib.key\_quit** is the key for leaving the program without validating user input. The default key is **CTRL\_\\**.



Type: CHAR(32)

**Related reference**

“EGL library ConsoleLib” on page 886

**lastKeyTyped()**

The system function **ConsoleLib.lastKeyTyped** returns the integer code of the last physical key that was pressed on the keyboard.

```
ConsoleLib.lastKeyTyped( )  
returns (result INT)
```

*result*

An integer that represents the last key pressed.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library ConsoleLib” on page 886

**menuLine**

The system variable **ConsoleLib.menuLine** contains the line location where menus are displayed in a window. It affects the properties of windows when they are opened.

Type: INT

**Related reference**

“EGL library ConsoleLib” on page 886

**messageLine**

The system variable **ConsoleLib.messageLine** is the window location where messages are displayed.

Type: INT

**Related reference**

“EGL library ConsoleLib” on page 886

**messageResource**

The system variable **ConsoleLib.messageResource** is the file name of the resource bundle from which help and other messages are loaded. If this variable has no value, the EGL runtime inspects the file identified in the Java runtime property **vgj.messages.file**.

Type: CHAR(255)

**Related concepts**

“Syntax diagram for EGL functions” on page 884

“Console user interface” on page 207

**Related reference**

“ConsoleUI parts and related variables” on page 209

“EGL library ConsoleLib” on page 886  
“Java runtime properties (details)” on page 642

## nextField()

The system function **ConsoleLib.nextField** moves the cursor to the next form field according the defined field travel order. This function is valid in an **openUI** statement that acts on a console form.

```
ConsoleLib.nextField( )
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## openWindow()

The system function **ConsoleLib.openWindow** makes a window visible, adds it to the top of the window stack.

```
ConsoleLib.openWindow(window1 Window inOut)
```

*window1*

A variable of type Window.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## openWindowByName()

The system function **ConsoleLib.openWindowByName** makes a window visible and adds it to the top of the window stack.

```
ConsoleLib.openWindowByName(name STRING in)
```

*name*

The value of the Window name field.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library ConsoleLib” on page 886

## openWindowWithForm()

The system function **ConsoleLib.openWindowWithForm** makes a window visible, adds it to the top of the window stack and displays the form in the window. The window is re-sized to fit the form.

```
ConsoleLib.openWindowWithForm(  
  window1 Window inOut,  
  form ConsoleForm in)
```

*window1*

A variable of type Window.

*form*

A variable of type ConsoleForm.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### openWindowWithFormByName()

The system function **ConsoleLib.openWindowWithFormByName** activates a window, makes it visible, and displays the specified console form. The window is re-sized to fit the form.

```
ConsoleLib.openWindowWithFormByName(  
    windowName STRING in,  
    formName STRING in)
```

*windowName*

The value of the Window name field.

*formName*

The value of the ConsoleForm name field.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### previousField()

The system function **ConsoleLib.previousField** moves the cursor to the previous form field according to the defined field tab order.

```
ConsoleLib.previousField( )
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### promptLine

The system variable **ConsoleLib.promptLine** contains the number of the line at which prompts are displayed in a window.

Type: INT

#### Related reference

“EGL library ConsoleLib” on page 886

### promptLineMode()

The system function **ConsoleLib.promptLineMode** displays the string in line mode and waits for user input, which is submitted when the user presses **Enter**.

```
ConsoleLib.promptLineMode(message String in)  
returns (result STRING)
```

*result*

The user input.

*message*

The phrase to display.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **quitRequested**

The system variable **ConsoleLib.quitRequested** indicates that a **QUIT** signal has been received (or simulated). If **true**, an **QUIT** signal has been received. If **false**, a **QUIT** signal has not been received.

Type: Boolean

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **screen**

The system variable **ConsoleLib.screen** automatically defines a default, borderless window. The dimensions of the screen are equal to the dimensions of the available display surface.

Type: Window

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **scrollDownLines()**

The system function **ConsoleLib.scrollDownLines** scrolls the on-screen data toward the bottom of the data by the specified number of lines.

**ConsoleLib.scrollDownLines**(*numLines* INT *in*)

*numLines*

The number of lines to scroll downward.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“EGL library ConsoleLib” on page 886

### **scrollDownPage()**

The system function **ConsoleLib.scrollDownPage** scrolls the on-screen data one page toward the bottom of the data.

**ConsoleLib.scrollDownPage**( )

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### scrollUpLines()

The system function **ConsoleLib.scrollUpLines** scrolls the on-screen data toward the top of the data by the specified number of lines.

**ConsoleLib.scrollUpLines**(*numLines* INT in)

*numLines*

The number of lines to scroll up.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### scrollUpPage()

The system function **ConsoleLib.scrollUpPage** scrolls the on-screen data by one page toward the top of the data.

**ConsoleLib.scrollUpPage**( )

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### setArrayLine()

The system function **ConsoleLib.setArrayLine** moves the selection to the specified program record. If necessary, the data is scrolled to make the selected record visible.

**ConsoleLib.setArrayLine**(*recordNumber* INT in)

*recordNumber*

The record to select.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### setCurrentArrayCount()

The system function **ConsoleLib.setCurrentArrayCount** specifies how many rows exist in a dynamic array that is bound to an on-screen arrayDictionary. This function is useful only if you invoke it before issuing the **openUI** statement that uses the arrayDictionary.

**ConsoleLib.setCurrentArrayCount**(*count* INT in)

*count*

The number of array entries when the openUI statement begins.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

“openUI” on page 726

### showAllMenuItems()

The system function **ConsoleLib.showAllMenuItems** shows all menu items in the currently displayed menu.

```
ConsoleLib.showAllMenuItems( )
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### showHelp()

The system function **ConsoleLib.showHelp** displays a help message. The string argument is the key for the message in the resource bundle configured with the **ConsoleLib.messageResource** field.

```
ConsoleLib.showHelp(helpKey STRING in)
```

*helpKey*

The key that looks up the text for a help message.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### showMenuItem()

The system function **ConsoleLib.showMenuItem** shows the specified menu item so that it can be selected by the user. By default all menu items are shown.

```
ConsoleLib.showMenuItem(item MenuItem in)
```

*item*

The menu item to show.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

### showMenuItemByName

The system function **ConsoleLib.showMenuItemByName** shows the specified menu item so that it can be selected by the user. By default all menu items are shown.

```
ConsoleLib.showMenuItemByName(name STRING in)
```

*name*

The value of the MenuItem name field.

#### Related reference

“EGL library ConsoleLib” on page 886

### sqlInterrupt

For the system variable **ConsoleLib.sqlInterrupt**, if **yes**, the user can interrupt SQL statements being processed. If **no**, the user can only interrupt **OpenUI** statements. Variable **sqlInterrupt** is used in combination with the *deferInterrupt* and *deferQuit* variables.

Type: Boolean

#### Related reference

“EGL library ConsoleLib” on page 886

### updateWindowAttributes()

The system function **updateWindowAttributes()** reads the global value matching the WindowAttributeKind value(s), and updates the current active window with this value. If the WindowAttributeKind value is one of Color, Intensity, or Highlight, all open windows will have their values updated, not just the active window.

This function is required for Informix 4GL compatibility. New EGL users should not need this function, as they can get the same results more easily through other means, such as setting attributes before opening a window.

```
ConsoleLib.updateWindowAttributes(  
    attribute WindowAttributeKind in)
```

*attribute*

One or more values of the enumeration WindowAttributeKind. For a list of those values, see *Enumerations in EGL*.

*form*

A variable of type ConsoleForm.

#### Related concepts

“Enumerations in EGL” on page 578

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library ConsoleLib” on page 886

## EGL library ConverseLib

The Converse library provides the functions shown in the table below.

Function	Description
clearScreen ()	Clears the screen, as is useful before the program issues a converse statement in a text application.
displayMsgNum ( <i>msgNumber</i> )	Retrieves a value from the program’s message table. The message is presented the next time that a form is presented by a <b>converse</b> , <b>display</b> , <b>print</b> , or <b>show</b> statement.

Function	Description
<code>result = fieldInputLength (textField)</code>	Returns the number of characters that the user typed in the input field when the text form was last presented. That number does not include leading or trailing blanks or nulls.
<code>pageEject ()</code>	Advances print-form output to the top of the next page, as is useful before the program issues a print statement.
<code>validationFailed (msgNumber)</code>	<ul style="list-style-type: none"> <li>If invoked in a field-validation function in a text application, <b>ConverseLib.validationFailed</b> causes the re-presentation of the received text form after all validation functions are processed. The last-invoked <b>ConverseLib.validationFailed</b> determines what message is displayed.</li> <li>If invoked outside a validation function, <b>ConverseLib.validationFailed</b> presents the specified message the next time that a form is presented by a <b>converse</b>, <b>display</b>, <b>print</b>, or <b>show</b> statement. The behavior in this case is like that of <b>ConverseLib.displayMsgNum</b>.</li> </ul>

## clearScreen()

The system function **ConverseLib.clearScreen** clears the screen, as is useful before the program issues a converse statement in a text application.

**ConverseLib.clearScreen( )**

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“converse” on page 672

“EGL library ConverseLib” on page 916

## displayMsgNum()

The system function **ConverseLib.displayMsgNum** retrieves a value from the program’s message table. The message is presented the next time that a form is presented by a **converse**, **display**, **print**, or **show** statement.

If possible, the message presentation is on the form itself, in the field to which the form property **msgField** refers. If the form property **msgField** has no value, the message is displayed previous to the display of the form, on a separate, modal screen or on a printable page.

**ConverseLib.displayMsgNum** takes as its only argument a value that is compared against each cell in the first column of the program’s *message table*, which is the data table to which the program’s **msgTablePrefix** property refers. The message retrieved by that function is in the second column of the same row.

**ConverseLib.displayMsgNum(msgNumber INT in)**



*msgNumber*

The message is retrieved from the message table by number. The argument must be an integer literal or an item of primitive type SMALLINT or INT or the BIN equivalent.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library ConverseLib” on page 916

## **fieldInputLength()**

The system function **ConverseLib.fieldInputLength** returns the number of characters that the user typed in the input field when the text form was last presented. That number does not include leading or trailing blanks or nulls.

If the field is at its originally defined state, the function returns a length of 0. For example, if the field contains the *value* property and it has not been modified during execution in any way, then the length is calculated as 0. The *set form initial* statement resets the field to its originally defined state. If the field is not at its originally defined state, then the length is calculated based on what was displayed or entered on the last converse statement.

```
ConverseLib.fieldInputLength(textField TestFormField in)  
returns(result INT)
```

*result*

The number of characters that the user typed.

*textField*

The name of the text field.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library ConverseLib” on page 916

## **pageEject()**

The system function **ConverseLib.pageEject** advances print-form output to the top of the next page, as is useful before the program issues a print statement.

```
ConverseLib.pageEject( )
```

For other details on printing, see *Print forms*.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

“Print forms” on page 186

**Related reference**

“EGL library ConverseLib” on page 916

“print” on page 737

## **validationFailed()**

The system function **ConverseLib.validationFailed** is used in either of two ways:

- If invoked in a field-validation function in a text application, **ConverseLib.validationFailed** causes the re-presentation of the received text form after all validation functions are processed. The last-invoked **ConverseLib.validationFailed** determines what message is displayed.  
If possible, the message presentation is on the form itself, in the field to which the form property **msgField** refers. If the form property **msgField** has no value, the message is displayed previous to the display of the form, on a separate, modal screen.
- If invoked outside a validation function, **ConverseLib.validationFailed** presents the specified message the next time that a form is presented by a **converse**, **display**, **print**, or **show** statement. The behavior in this case is like that of **ConverseLib.displayMsgNum**.

In any case, the value assigned to **ConverseLib.validationFailed** is stored in the system variable **ConverseVar.validationMsgNum**.

**ConverseLib.validationFailed**(*[msgNumber* **INT** *in]*)

*msgNumber*

The number of the message to display. The argument must be an integer literal or an item of primitive type SMALLINT or INT or the BIN equivalent. This number is compared against each cell in the first column of the program's *message table*, which is the data table to which the program's **msgTablePrefix** property refers. The retrieved message is in the second column of the same row.

The message number is 9999 by default.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"displayMsgNum()" on page 917

"validationMsgNum" on page 1061

"EGL library ConverseLib" on page 916

## EGL library DateTimeLib

The date-and-time system variables let you retrieve the system date and time in a variety of formats, as shown in the next table.

System variable	Description
<i>result</i> = currentDate ()	Contains the current system date in eight-digit Gregorian format (yyyyMMdd); you can assign this system variable to a variable of type DATE.
<i>result</i> = currentTime ()	Contains the current system time in six-digit format (HHmmss); you can assign this system variable to a variable of type TIME.
<i>result</i> = currentTimeStamp ()	Contains the current system time and date as a timestamp in twenty-digit Julian format (yyyyMMddHHmmssffffff); you can assign this system variable to a variable of type TIMESTAMP.
<i>result</i> = dateOf ( <i>aTimeStamp</i> )	Returns a date derived from a variable of type TIMESTAMP.

System variable	Description
<i>result</i> = <i>dateValue</i> ( <i>dateAsString</i> )	Returns a DATE value that corresponds to an input string.
<i>result</i> = <i>dateValueFromGregorian</i> ( <i>gregorianIntegerDate</i> )	Returns a DATE value that corresponds to an integer representation of a Gregorian date.
<i>result</i> = <i>dateValueFromJulian</i> ( <i>julianIntegerDate</i> )	Returns a DATE value that corresponds to an integer representation of a Julian date.
<i>result</i> = <i>dayOf</i> ( <i>aTimeStamp</i> )	Returns a positive integer that represents a day of the month, as derived from a variable of type TIMESTAMP.
<i>result</i> = <i>extend</i> ( <i>extensionField</i> [, <i>mask</i> ])	Converts a timestamp, time, or date into a longer or shorter timestamp value.
<i>result</i> = <i>intervalValue</i> ( <i>intervalAsString</i> )	Returns an INTERVAL value that reflects a string constant or literal.
<i>result</i> = <i>intervalValueWithPattern</i> ( <i>intervalAsString</i> [, <i>intervalMask</i> ])	Returns an INTERVAL value that reflects a string constant or literal and is built based on an interval mask that you specify.
<i>result</i> = <i>mdy</i> ( <i>month</i> , <i>day</i> , <i>year</i> )	Returns a DATE value derived from three integers that represent the month, day of the month, and year of a calendar date.
<i>result</i> = <i>monthOf</i> ( <i>aTimeStamp</i> )	Returns a positive integer that represents a month, as derived from a variable of type TIMESTAMP.
<i>result</i> = <i>timeOf</i> ([ <i>aTimeStamp</i> ])	Returns a string that represents the time of day derived from either a TIMESTAMP variable or the system clock.
<i>result</i> = <i>timestampFrom</i> ( <i>tsDate</i> <i>tsTime</i> )	Contains the current system time and date as a timestamp in twenty-digit Julian format (yyyyMMddHHmmssffffff); you can assign this system variable to a variable of type TIMESTAMP.
<i>result</i> = <i>timestampValue</i> ( <i>timestampAsString</i> )	Returns a TIMESTAMP value that reflects a string constant or literal.
<i>result</i> = <i>timestampValueWithPattern</i> ( <i>timestampAsString</i> [, <i>timestampMask</i> ])	Returns a TIMESTAMP value that reflects a string and is built based on a timestamp mask that you specify.
<i>result</i> = <i>timeValue</i> ( <i>timeAsString</i> )	Returns a TIME value that reflects a string constant or literal.
<i>result</i> = <i>weekdayOf</i> ( <i>aTimeStamp</i> )	Returns a positive integer (0-6) that represents a day of the week, as derived from a variable of type TIMESTAMP.
<i>result</i> = <i>yearOf</i> ( <i>aTimeStamp</i> )	Returns an integer that represents a year, as derived from a variable of type TIMESTAMP.

To set a date, time, or timestamp variable, you can assign `VGVar.currentGregorianDate`, `DateTimeLib.currentTime`, and `DateTimeLib.currentTimeStamp`, respectively. The functions that return formatted character text cannot be used for this purpose.

#### Related reference

“EGL statements” on page 88

### currentDate()

The system function **DateTimeLib.currentDate** reads the system clock and returns a DATE value that represents the current calendar date. The function returns only the current date, not the time of day.

```
DateTimeLib.currentDate( )  
returns (result DATE)
```

*result*

A DATE value that represents the current calendar date.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“DATE” on page 41

“EGL library DateTimeLib” on page 919

### currentTime()

The system function **DateTimeLib.currentTime** retrieves the current system time in six-digit format (HHmmss). The value is automatically updated each time it is referenced by your program.

```
DateTimeLib.currentTime( )  
returns (result TIME)
```

*result*

A TIME value that represents the current system time.

You can use **DateTimeLib.currentTime** in these ways:

- As the source in an assignment or **move** statement
- As the argument in a **return** statement

The characteristics of **DateTimeLib.currentTime** are as follows:

#### Primitive type

TIME

#### Data length

6

#### Value saved across segments

No

#### Example:

```
myTime = DateTimeLib.currentTime;
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library DateTimeLib” on page 919

### currentTimeStamp()

The system function **DateTimeLib.currentTimeStamp** retrieves the current system time and date as a timestamp in twenty-digit format (yyyyMMddHHmmssffffff). The value is automatically updated each time it is referenced by your program.

**DateTimeLib.currentTimeStamp( )**  
returns (*result* **TIMESTAMP**)

*result*

A **TIMESTAMP** value that represents the current system time and date.

You can use **DateTimeLib.currentTimeStamp** in these ways:

- As the source in an assignment or **move** statement
- As the argument in a **return** statement

The characteristics of **DateTimeLib.currentTimeStamp** are as follows:

**Primitive type**

**TIMESTAMP**

**Data length**

20

**Value saved across segments**

No

**Example:**

```
myTimeStamp = DateTimeLib.currentTimeStamp;
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library DateTimeLib” on page 919

**dateOf()**

The system function **DateTimeLib.dateOf** returns a **DATE** value derived from a variable of type **TIMESTAMP**.

**DateTimeLib.dateOf**(*aTimeStamp* **TIMESTAMP** in)  
returns (*result* **DATE**)

*result*

A **DATE** value.

*aTimeStamp*

The value from which the date is derived.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“DATE” on page 41

“EGL library DateTimeLib” on page 919

**dateValue()**

The function **DateTimeLib.dateValue** returns a **DATE** value that corresponds to a string.

**DateTimeLib.dateValue**(*dateAsString* **STRING** in)  
returns (*result* **DATE**)

*result*

A variable of type **DATE**.

*dateAsString*

A string constant or literal containing digits that reflect the mask "yyyyMMdd". For details, see *DATE*.

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"DATE" on page 41

"Datetime expressions" on page 591

**dateValueFromGregorian()**

The function **DateTimeLib.dateValueFromGregorian** returns a DATE value that corresponds to an integer representation of a Gregorian date.

```
DateTimeLib.dateValueFromGregorian(  
    gregorianIntegerDate INT in)  
returns (result DATE)
```

*result*

A variable of type DATE.

*gregorianIntegerDate*

An integer value representing a Gregorian date in the format 00YYMMDD or 00YYMMDD.

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"DATE" on page 41

"EGL library DateTimeLib" on page 919

**dateValueFromJulian()**

The function **DateTimeLib.dateValueFromJulian** returns a DATE value that corresponds to an integer representation of a Julian date.

```
DateTimeLib.dateValueFromJulian(  
    julianIntegerDate INT in)  
returns (result DATE)
```

*result*

A variable of type DATE.

*julianIntegerDate*

A integer value representing a Julian date in the format 00YYYYDDD or 00YYDD.

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"DATE" on page 41

"EGL library DateTimeLib" on page 919

**dayOf()**

The system function **DateTimeLib.dayOf** returns a positive integer that represents a day (1-7), as derived from a variable of type *TIMESTAMP*.

```
DateTimeLib.dayOf(aTimeStamp TIMESTAMP in)  
returns (result INT)
```

*result*

A positive integer that corresponds to the day of the month.

*aTimeStamp*

The variable from which the day is derived.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“DATE” on page 41

“EGL library DateTimeLib” on page 919

### extend()

The system function **DateTimeLib.extend** converts a timestamp, time, or date into a longer or shorter timestamp value. Examples are as follows:

- If you have an input timestamp defined as “ddHH” (day and hour) and provide a timestamp mask of “ddHHmm” (day, hour, and minute), **DateTimeLib.extend** returns an extended value that matches the mask
- If you have an input timestamp defined as “yyyyMMddHHmmss” (year, month, day, hour, minute, and second) and provide a timestamp mask “yyyy” (year), **DateTimeLib.extend** returns a shortened value that matches the mask

```
DateTimeLib.extend(  
    extensionField dateOrTimeOrTimeStamp in in  
    [, mask outputTimeStampMask in in  
    ])  
returns (result TIMESTAMP)
```

*result*

A variable of type TIMESTAMP.

*extensionField*

The name of a field of type TIMESTAMP, TIME, or DATE. The field contains the value to be extended or shortened.

*mask*

A string literal or constant that defines the mask of the timestamp value returned by the function. For details, see *TIMESTAMP*.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“Datetime expressions” on page 591

### intervalValue()

The datetime value function **DateTimeLib.intervalValue** returns an INTERVAL value that reflects a string constant or literal and is built based on the default interval mask, which is *yyyyMM*.

The input string must contain six digits. The first four digits represent the number of years in the interval, and the last two represent the number of months.

If you wish to specify a mask other than *yyyyMM*, invoke **DateTimeLib.intervalValueWithPattern**.

```
DateTimeLib.intervalValue(intervalAsString STRING in)  
returns (result INTERVAL)
```

*result*

A variable of type INTERVAL

*intervalAsString*

A string constant or literal that contains six digits whose meaning is indicated by the interval mask *yyyyMM*

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“Datetime expressions” on page 591

“INTERVAL” on page 42

“intervalValueWithPattern()”

### intervalValueWithPattern()

The datetime value function **DateTimeLib.intervalValueWithPattern** returns an INTERVAL value that reflects a string constant or literal and (optionally) is built based on an interval mask that you specify. If the mask is *yyyy*, for example, the input string must contain four digits, and those digits represent the number of years represented in the interval.

```
DateTimeLib.intervalValueWithPattern(  
    intervalAsString STRING in  
    [, intervalMask STRING in  
    ]  
)  
returns (result INTERVAL)
```

*result*

A variable of type INTERVAL.

*intervalAsString*

A string constant or literal that contains digits whose meaning is indicated by the interval mask.

*intervalMask*

Specifies an interval mask that gives meaning to each digit in the first parameter. The default mask is *yyyyMM*. For other details, see *INTERVAL*.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“Datetime expressions” on page 591

“INTERVAL” on page 42

### mdy()

The mdy operator returns a DATE value derived from three integers that represent the month, day of the month, and year of a calendar date.

```
DateTimeLib.mdy(  
    month INT in,  
    day INT in,  
    year INT in)  
returns (result DATE)
```

*result*

A DATE value.

*month*

An integer in the range 1 through 12, representing the month.



*day*

An integer representing the day of the month in the range 1 through 28, 29, 30, or 31, depending on the month.

*year*

A four-digit integer representing the year.

An error results if you specify values outside the range of days and months in the calendar or if the number of operands is not three. You must enclose the three integer expression operands between parentheses, separated by commas, just as you would if MDY( ) were a function. The third expression cannot be the abbreviation for the year. For example, 99 specifies a year in the first century, approximately 1,900 years ago.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“DATE” on page 41

“EGL library DateTimeLib” on page 919

### **monthOf()**

The system function **DateTimeLib.monthOf** returns a positive integer that represents a month, as derived from a variable of type **TIMESTAMP**.

```
DateTimeLib.monthOf(aTimeStamp TIMESTAMP in)  
returns (result INT)
```

*result*

A positive integer that represents a month.

*aTimeStamp*

The **TIMESTAMP** variable from which the month is derived.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“DATE” on page 41

“EGL library DateTimeLib” on page 919

### **timeOf()**

The system function **DateTimeLib.timeOf** returns a string that represents the time of day derived from either a **TIMESTAMP** variable or the system clock.

```
DateTimeLib.timeOf([aTimeStamp TIMESTAMP in])  
returns (result STRING)
```

*result*

The time-of-day portion of the *aTimeStamp* argument, as based on a 24-hour clock and the following format:

hh:mm:ss

*hh* The hour as a two-digit string.

*mm*

The minute as a two-digit string.

*ss* The second as a two-digit string.

*aTimeStamp*

A DATETIME value. If no value is specified, the operator returns a character string representing the current time

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“DATE” on page 41

“EGL library DateTimeLib” on page 919

### timestampFrom()

The function **DateTimeLib.timestampFrom** returns a TIMESTAMP value that is built based on a DATE and TIME that you specify.

```
DateTimeLib.timestampFrom(  
    tsDate DATE in,  
    tsTime TIME in)  
returns (result TIMESTAMP)
```

*result*

A value of type TIMESTAMP.

*tsDate*

A variable of type DATE.

*tsTime*

A variable of type TIME.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“Datetime expressions” on page 591

“EGL library DateTimeLib” on page 919

“TIMESTAMP” on page 44

### timestampValue()

The function **DateTimeLib.timestampValue** returns a TIMESTAMP value that reflects a string constant or literal and is built based on the default timestamp mask, which is *yyyyMMddHHmmss*.

The input string must contain fourteen digits:

- The first four digits represent the year
- The next two represent the numeric month
- The next two represent the day of the month
- The next two represent the number of hours (from 00 to 24)
- The next two represent the number of minutes within the hour
- The last two represent the number of seconds within the minute

If you wish to specify a mask other than *yyyyMMddHHmmss*, invoke **DateTimeLib.timestampValueWithPattern**.

```
DateTimeLib.timestampValue(timestampAsString STRING in)  
returns (result TIMESTAMP)
```

*result*

A variable of type TIMESTAMP.

*timestampAsString*

A string constant or literal that contains fourteen digits whose meaning is indicated by the timestamp mask *yyyyMMddHHmmss*

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"Datetime expressions" on page 591

"TIMESTAMP" on page 44

"timestampValueWithPattern()"

### timestampValueWithPattern()

The function **DateTimeLib.timestampValueWithPattern** returns a **TIMESTAMP** value that reflects a string constant or literal and (optionally) is built based on a timestamp mask that you specify. If the mask is "yyyy", for example, the input string must contain four digits, and those digits represent the year value in the timestamp.

```
DateTimeLib.timestampValueWithPattern(  
    timestampAsString STRING in  
    [, timestampMask STRING in  
    ]  
)  
returns (result TIMESTAMP)
```

*result*

A variable of type **TIMESTAMP**.

*timestampAsString*

A string constant or literal that contains digits whose meaning is indicated by the timestamp mask.

*timestampMask*

Specifies a timestamp mask that gives meaning to each digit in the first parameter. The default mask is *yyyyMMddHHmmss*. For other details, see *TIMESTAMP*.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"Datetime expressions" on page 591

"TIMESTAMP" on page 44

### timeValue()

The datetime value function **DateTimeLib.timeValue** returns a **TIME** value that reflects a string constant or literal.

```
DateTimeLib.timeValue(timeAsString STRING in)  
returns (result TIME)
```

*result*

A variable of type **TIME**.

*timeAsString*

A string constant or literal containing digits that reflect the mask "HHmmss". For details, see *TIME*.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

“Datetime expressions” on page 591

“TIME” on page 44

### weekdayOf()

The system function **DateTimeLib.weekdayOf** returns a positive integer that represents a day of the week, as derived from a variable of type **TIMESTAMP**. The number 0 represents Sunday, 1 represents Monday, and so on.

**DateTimeLib.weekdayOf**(*aTimeStamp* **TIMESTAMP** in)  
returns (*result* **INT**)

*result*

A positive integer from 0 to 6.

*aTimeStamp*

The **TIMESTAMP** variable from which the day is derived.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“DATE” on page 41

“EGL library DateTimeLib” on page 919

### yearOf()

The system function **DateTimeLib.yearOf** returns a four-digit integer that represents a year, as derived from a variable to type **TIMESTAMP**.

**DateTimeLib.yearOf**(*aTimeStamp* **TIMESTAMP** in)  
returns (*result* **INT**)

*result*

The integer that represents the year.

*aTimeStamp*

The **TIMESTAMP** variable from which the year is derived.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“DATE” on page 41

“EGL library DateTimeLib” on page 919

## EGL library DLILib

The next table shows the system functions in the library **DLILib** and is followed by a table that shows the one variable in that library.

System function and invocation	Description
“EGLTDLI()” on page 930 ( <i>func, pcbrecord, parms...</i> )	Invokes a DL/I function directly using the CBLTDLI interface.
“AIBTDLI()” on page 930 ( <i>func, pcbrecord, parms...</i> )	Invokes a DL/I function directly using the AIBTDLI interface.

The EGL library **VGLib** contains the function **VGLib.VGTDLI()**, which differs from **DLILib.EGLTDLI()** only in that it accesses the PCB with an indexed variable.

The next table shows the one system variable in the library **DLILib**.

System variable	Description
"psbData" on page 931	Holds the name of the runtime PSB.

#### Related reference

"EGLTDLI()"

### AIBTDLI()

The system function **AIBTDLI()** uses the AIBTDLI interface to invoke a DL/I function directly.

```
DLILib.AIBTDLI() (  
  func CHAR(4) in,  
  pcbrecord DB_PCBRecord in  
  parms... ANY in)
```

*func*

A four-character DL/I function name such as ISRT or GHNP

*pcbrecord*

The name you assigned to a PCB record within a PSB record in your program

*parms...*

A complete list of parameters, matching in number and type those that the given DL/I function requires

To invoke a DL/I function using the CBLTDLI interface, use **DLILib.EGLTDLI()** or **VGLib.VGTDLI()**.

#### Related reference

"EGL library DLILib" on page 929

"EGLTDLI()"

"EGLTDLI()"

### EGLTDLI()

The system function **EGLTDLI()** uses the CBLTDLI interface to invoke a DL/I function directly.

```
DLILib.EGLTDLI() (  
  func CHAR(4) in,  
  pcbrecord DB_PCBRecord in  
  parms... ANY in)
```

*func*

A four-character DL/I function name such as ISRT or GHNP

*pcbrecord*

The name you assigned to a PCB record within a PSB record in your program

*parms...*

A complete list of parameters, matching in number and type those that the given DL/I function requires

To invoke a DL/I function using the AIBTDLI interface, use **DLILib.AIBTDLI()**.

#### Related reference

"EGL library DLILib" on page 929

"AIBTDLI()" on page 930

"EGLTDLI()" on page 930

### psbData

The system variable **DLILib.psbData** contains both the name of the runtime PSB and an address with which that PSB is accessed.

The variable is based on the predefined record part PSBDataRecord, which has the following structure:

```
Record PSBDataRecord
  psbName char(8);
  psbRef int;
end
```

If your program switches from one PSB to another (as is possible outside of IMS), you can set and test the field **DLILib.psbData.psbName** directly. Never update the field **DLILib.psbData.psbRef**, which contains an address.

The initial value of **DLILib.psbData.psbName** is the value in the predefined field **defaultPSBName**. That predefined field is in the PSB record that is assigned to the program property **psb**. If you do not set that field, its value is the name of the PSBRecord part on which the PSB record is based.

#### Related reference

"EGL library DLILib" on page 929

## EGL library J2EELib

The next table lists the system functions in the library J2EELib.

Function	Description
<code>clearRequestAttr (key)</code>	Removes the argument that is associated with the specified key in the request object.
<code>clearSessionAttr (key)</code>	Removes the argument that is associated with the specified key in the session object.
<code>getRequestAttr (key, argument)</code>	Uses a specified key to retrieve an argument from the request object into a specified variable.
<code>getSessionAttr (key, argument)</code>	Uses a specified key to retrieve an argument from the session object into a specified variable.
<code>setRequestAttr (key, argument)</code>	Uses a specified key to place a specified argument in the request object.
<code>setSessionAttr (key, argument)</code>	Uses a specified key to place a specified argument in the session object.

### clearRequestAttr()

The system function **J2EELib.clearRequestAttr** removes the argument that is associated with the specified key in the request object. This function is useful in PageHandlers and in programs that run in Web applications.

You can set an argument in the request object by using the system function `J2EELib.setRequestAttr`. You can retrieve the argument by using the system function `J2EELib.getRequestAttr`.

**J2EELib.clearRequestAttr**(*key* **STRING** in)

*key*

A string literal or an expression of type `String`

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"PageHandler" on page 223

#### **Related reference**

"EGL library J2EELib" on page 931

"getRequestAttr()"

"setRequestAttr()" on page 933

### **clearSessionAttr()**

The system function **J2EELib.clearSessionAttr** removes the argument that is associated with the specified key in the session object. This function is useful in `PageHandlers` and in programs that run in Web applications.

You can set an argument in the session object by using the system function `J2EELib.setSessionAttr`. You can retrieve the argument by using the system function `J2EELib.getSessionAttr`.

**J2EELib.clearSessionAttr**(*key* **STRING** in)

*key*

A string literal or an expression of type `STRING`

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"PageHandler" on page 223

#### **Related reference**

"EGL library J2EELib" on page 931

"getSessionAttr()" on page 933

"setSessionAttr()" on page 934

### **getRequestAttr()**

The system function **J2EELib.getRequestAttr** uses a specified key to retrieve an argument from the request object into a specified variable. This function is useful in `PageHandlers` and in programs that run in Web applications.

If an object is not found with the specified key, the target variable is unchanged. If the retrieved object is of the wrong type, an exception is thrown and the program or `PageHandler` terminates.

You can place an argument in the request object by using the system function **J2EELib.setRequestAttr**. The argument object placed in the servlet's request collection is available for access as long as the servlet request is valid. Submitting a form from a page causes the creation of a new request.

**J2EELib.getRequestAttr**(  
  *key* **STRING** in,  
  *argument attribute* inOut)

*key*

A character literal or an item of any character type.

*argument*

An item, record, or array.

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"PageHandler" on page 223

#### **Related reference**

"EGL library J2EELib" on page 931

"setRequestAttr()"

### **getSessionAttr()**

The system function **J2EELib.getSessionAttr** uses a specified key to retrieve an argument from the session object into a specified variable. This function is useful in PageHandlers and in programs that run in Web applications.

If an object is not found with the specified key, the target variable is unchanged. If the retrieved object is of the wrong type, an exception is thrown and the program or PageHandler terminates.

You can place an argument in the session object by using the system function **J2EELib.setSessionAttr**.

```
J2EELib.getSessionAttr(  
    key STRING in,  
    argument attribute in)
```

*key*

A character literal or an item of any character type.

*argument*

An item, record, or array.

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"PageHandler" on page 223

#### **Related reference**

"EGL library J2EELib" on page 931

"setSessionAttr()" on page 934

### **setRequestAttr()**

The system function **J2EELib.setRequestAttr** uses a specified key to place a specified argument in the request object. This function is useful in PageHandlers and in programs that run in Web applications. You can retrieve the argument later by using the system function **J2EELib.getRequestAttr**.

```
J2EELib.setRequestAttr(  
    key STRING in,  
    argument attribute in)
```

*key*

A character literal or an item of any character type.

*argument*

An item, record, or array.



In the generated Java output, item arguments are passed as primitive Java objects (String, Integer, Decimal, and so on). Record arguments are passed as record beans. Arrays are passed as an array list of the associated type. The argument object is placed in the servlet's request collection and is available for access as long as the servlet request is valid. Submitting a form from a page causes the creation of a new request.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

"PageHandler" on page 223

#### Related reference

"EGL library J2EELib" on page 931

"getRequestAttr()" on page 932

### setSessionAttr()

The system function **J2EELib.setSessionAttr** uses a specified key to place a specified argument in the session object. This function is useful in PageHandlers and in programs that run in Web applications. You can retrieve the argument later by using the system function **J2EELib.getSessionAttr**.

```
J2EELib.setSessionAttr(  
    key STRING in,  
    argument attribute in)
```

*key*

A character literal or an item of any character type.

*argument*

An item, record, or array.

In the generated Java output, item arguments are passed as primitive Java objects (String, Integer, Decimal, and so on). Record arguments are passed as record beans. Arrays are passed as an array list of the associated type.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

"PageHandler" on page 223

#### Related reference

"EGL library J2EELib" on page 931

"getSessionAttr()" on page 933

## EGL library JavaLib

The Java access functions are listed in the table.

Function	Description
<i>result</i> = getField ( <i>identifierOrClass</i> , <i>field</i> )	Returns the value of a specified field of a specified object or class
<i>result</i> = invoke ( <i>identifierOrClass</i> , <i>method</i> [, <i>argument</i> ])	Invokes a method on a Java object or class and may return a value
<i>result</i> = isNull ( <i>identifier</i> )	Returns a value (1 for true, 0 for false) to indicate whether a specified identifier refers to a null object

Function	Description
<i>result</i> = isObjID ( <i>identifier</i> )	Returns a value (1 for true, 0 for false) to indicate whether a specified identifier is in the object space
<i>result</i> = qualifiedTypeName( <i>identifier</i> )	Returns the fully qualified name of the class of an object in the object space
remove ( <i>identifier</i> )	Removes the specified identifier from the object space and, if no other identifiers refer to the object, removes the object
removeAll ()	Removes all identifiers and objects from the object space
setField ( <i>identifierOrClass</i> , <i>field</i> , <i>value</i> )	Sets the value of a field in a Java object or class
store ( <i>storeId</i> , <i>identifierOrClass</i> , <i>method</i> { <i>,argument</i> })	Invokes a method and places the returned object (or null) into the object space, along with a specified identifier
storeCopy ( <i>sourceId</i> , <i>targetID</i> )	Creates a new identifier based on another in the object space, so that both refer to the same object
storeField ( <i>storeId</i> , <i>identifierOrClass</i> , <i>field</i> )	Places the value of a class field or object field into the object space
storeNew( <i>storeId</i> , <i>class</i> { <i>,argument</i> })	Invokes the constructor of a class and places the new object into the object space

## Java access functions

The *Java access functions* are EGL system functions that allow your generated Java code to access native Java objects and classes; specifically, to access the public methods, constructors, and fields of the native code.

This EGL feature is made possible at run time by the presence of the *EGL Java object space*, which is a set of names and the objects to which those names refer. A single object space is available to your generated program and to all generated Java code that your program calls locally, whether the calls are direct or by way of another local generated Java program, to any level of call. The object space is not available in any native Java code.

To store and retrieve objects in the object space, you invoke the Java access functions. Your invocations include use of identifiers, each of which is a string that is used to store an object or to match a name that already exists in the object space. When an identifier matches a name, your code can access the object associated with the name.

The next sections are as follows:

- “Mappings of EGL and Java types”
- “Examples” on page 937
- “Error handling” on page 940

**Mappings of EGL and Java types:** Each of the arguments you pass to a method (and each value that you assign to a field) is mapped to a Java object or primitive type. Items of EGL primitive type CHAR, for example, are passed as objects of the Java String class. A cast operator is provided for situations in which the mapping of EGL types to Java types is not sufficient.

When you specify a Java name, EGL strips single- and double-byte blanks from the beginning and end of the value, which is case sensitive. The truncation precedes any cast. This rule applies to string literals and to items of type CHAR, DBCHAR, MBCHAR, or UNICODE. No such truncation occurs when you specify either a method argument or field value (for example, the string " my data " is passed to a method as is), unless you cast the value to objID or null.

The next table describes all the valid mappings.

Category of Argument		Examples	Java Type
A string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE	No cast	"myString"	java.lang.String
	Cast with objId, which indicates an identifier	(objId)"myId"  x = "myId"; (objId)x	The class of the object to which the identifier refers
	Cast with null, as may be appropriate to provide a null reference to a fully qualified class	(null)"java.lang.Thread"  x = "java.util.HashMap"; (null)x	The specified class <b>Note:</b> You can't pass in a null-casted array such as (null)"int[]"
	Cast with char, which means that the first character of the value is passed (each example in the next column passes an "a")	(char)"abc"  x = "abc"; (char)x	char
An item of type FLOAT or a floating point literal	No cast	myFloatValue	double
An item of type HEX	No cast	myHexValue	byte array
An item of type SMALLFLOAT	No cast	mySmallFloat	float
An item of type DATE	No cast	myDate	java.sql.Date
An item of type TIME	No cast	myTime	java.sql.Time
An item of type TIMESTAMP	No cast	myTimeStamp	java.sql.Timestamp
An item of type INTERVAL	No cast	myInterval	java.lang.String
Floating point literal	No cast	-6.5231E96	double
Numeric item (or non-floating-point literal) that does not contain decimals; leading zeros are included in the number of digits for a literal	No cast, 1-4 digits	0100	short
	No cast, 5-9 digits	00100	int
	No cast, 9-18 digits	1234567890	long
	No cast, >18 digits	1234567890123456789	java.math.BigInteger

Category of Argument		Examples	Java Type
Numeric item Numeric item (or non-floating-point literal) that contains decimals; leading and trailing zeros are included in the number of digits for a literal	No cast, 1–6 digits	3.14159	float
	No cast, 7-18 digits	3.14159265	double
	No cast, >18 digits	56789543.222	java.math.BigDecimal
Numeric item or non-floating-point literal, with or without decimals	Cast with bigdecimal, biginteger, byte, double, float, short, int, long	X = 42;  (byte)X  (long)X	The specified primitive type; but if the value is out of range for that type, loss of precision occurs and the sign may change
	Cast with boolean, which means that non-zero is true, zero is false	X = 1; (boolean)X	boolean

**Note:** To avoid losing precision, use an EGL float item for a Java double, and an EGL smallfloat item for a Java float. Using one of the other EGL types will probably result in a value being rounded.

For details on the internal format of items in EGL, see the help pages on *Primitive types*.

**Examples:** This section gives examples on how to use Java access functions.

*Printing a date string:* The following example prints a date string:

```
// call the constructor of the Java Date class and
// assign the new object to the identifier "date".
JavaLib.storeNew( (objId)"date", "java.util.Date" );

// call the toString method of the new Date object
// and assign the output (today's date) to the charItem.
// In the absence of the cast (objId), "date"
// refers to a class rather than an object.
charItem = JavaLib.invoke( (objId)"date", "toString" );

// assign the standard output stream of the
// Java System class to the identifier "systemOut".
JavaLib.storeField( (objId)"systemOut",
    "java.lang.System", "out" );

// call the println method of the output
// stream and print today's date.
JavaLib.invoke( (objID)"systemOut","println",charItem );

// The use of "java.lang.System.out" as the first
// argument in the previous line would not have been
// valid, as the argument must either be a
// an identifier already in the object space or a class
// name. The argument cannot refer to a static field.
```

*Testing a system property:* The following example retrieves a system property and tests for the absence of a value:

```

// assign the name of an identifier to an item of type CHAR
valueID = "osNameProperty"

// place the value of property os.name into the
// object space, and relate that value (a Java String)
// to the identifier osNameProperty
JavaLib.store((objId)valueId, "java.lang.System",
    "getProperty", "os.name");

// test whether the property value is non-existent
// and process accordingly
myNullFlag = JavaLib.isNull( (objId)valueId );

if( myNullFlag == 1 )
    error = 27;
end

```

*Working with arrays:* When you work with Java arrays in EGL, use the Java class `java.lang.reflect.Array`, as shown in later examples and as described in the Java API documentation. You cannot use **JavaLib.storeNew** to create a Java array because Java arrays have no constructors.

You use the static method `newInstance` of `java.lang.reflect.Array` to create the array in the object space. After you create the array, you use other methods in that class to access the elements.

The method `newInstance` expects two arguments:

- A Class object that determines the type of array being created
- A number that specifies how many elements are in the array

The code that identifies the Class object varies according to whether you are creating an array of objects or an array of primitives. The subsequent code that interacts with the array also varies on the same basis.

*Working with an array of objects:* The following example shows how to create a 5-element object array that is accessible by use of the identifier "myArray":

```

// Get a reference to the class, for use with newInstance
JavaLib.store( (objId)"objectClass", "java.lang.Class",
    "forName", "java.lang.Object" );

// Create the array in the object space
JavaLib.store( (objId)"myArray", "java.lang.reflect.Array",
    "newInstance", (objId)"objectClass", 5 );

```

If you want to create an array that holds a different type of object, change the class name that is passed to the first invocation of **JavaLib.store**. To create an array of String objects, for example, pass "java.lang.String" instead of "java.lang.Object".

To access an element of an object array, use the `get` and `set` methods of `java.lang.reflect.Array`. In the following example, `i` and `length` are numeric items:

```

length = JavaLib.invoke( "java.lang.reflect.Array",
    "getLength", (objId)"myArray" );
i = 0;

while ( i < length )
    JavaLib.store( (objId)"element", "java.lang.reflect.Array",
        "get", (objId)"myArray", i );

// Here, process the element as appropriate

```

```

        JavaLib.invoke( "java.lang.reflect.Array", "set",
            (objId)"myArray", i, (objId)"element" );
        i = i + 1;
    end

```

The previous example is equivalent to the following Java code:

```

int length = myArray.length;

for ( int i = 0; i < length; i++ )
{
    Object element = myArray[i];

    // Here, process the element as appropriate

    myArray[i] = element;
}

```

*Working with an array of Java primitives:* To create an array that stores a Java primitive rather than an object, use a different mechanism in the steps that precede the use of `java.lang.reflect.Array`. In particular, obtain the `Class` argument to `newInstance` by accessing the static field `TYPE` of a primitive type class.

The following example creates `myArray2`, which is a 30-element array of integers:

```

// Get a reference to the class, for use with newInstance
JavaLib.storeField( (objId)"intClass",
    "java.lang.Integer", "TYPE");

// Create the array in the object space
JavaLib.store( (objId)"myArray2", "java.lang.reflect.Array",
    "newInstance", (objId)"intClass", 30 );

```

If you want to create an array that holds a different type of primitive, change the `Class` name that is passed to the invocation of **JavaLib.storeField**. To create an array of characters, for example, pass `"java.lang.Character"` instead of `"java.lang.Integer"`.

To access an element of an array of primitives, use the `java.lang.reflect.Array` methods that are specific to a primitive type. Such methods include `getInt`, `setInt`, `getFloat`, `setFloat`, and so forth. In the following example, `length`, `element`, and `i` are numeric items:

```

length = JavaLib.invoke( "java.lang.reflect.Array",
    "getLength", (objId)"myArray2" );
i = 0;

while ( i < length )
    element = JavaLib.invoke( "java.lang.reflect.Array",
        "getDouble", (objId)"myArray2", i );

    // Here, process an element as appropriate

    JavaLib.invoke( "java.lang.reflect.Array", "setDouble",
        (objId)"myArray2", i, element );
    i = i + 1;
end

```

The previous example is equivalent to the following Java code:

```

int length = myArray2.length;

for ( int i = 0; i < length; i++ )
{
    double element = myArray2[i];
}

```

```

        // Here, process an element as appropriate
        myArray2[i] = element;
    }

```

*Working with collections:* To iterate over a collection that is referenced by a variable called *list*, a Java program does as follows:

```

    Iterator contents = list.iterator();

    while( contents.hasNext() )
    {
        Object myObject = contents.next();
        // Process myObject
    }

```

Assume that `hasNext` is a numeric data and that your program related a collection to an identifier called *list*. The following EGL code is then equivalent to the Java code described earlier:

```

JavaLib.store( (objId)"contents", (objId)"list", "iterator" );
hasNext = JavaLib.invoke( (objId)"contents", "hasNext" );

while ( hasNext == 1 )
    JavaLib.store( (objId)"myObject", (objId)"contents", "next");

    // Process myObject
    hasNext = JavaLib.invoke( (objId)"contents", "hasNext" );
end

```

*Converting an array to a collection:* To create a collection from an array of objects, use the `asList` method of `java.util.Arrays`, as shown in the following example:

```

// Create a collection from array myArray
// and relate that collection to the identifier "list"
JavaLib.store( (objId)"list", "java.util.Arrays",
    "asList", (objId)"myArray" );

```

Next, iterate over *list*, as shown in the preceding section.

The transfer of an array to a collection works only with an array of objects, not with an array of Java primitives. Be careful not to confuse `java.util.Arrays` with `java.lang.reflect.Array`.

**Error handling:** Many of the Java access functions are associated with error codes, as described in the function-specific help pages. If the value of the system variable **VGVar.handleSysLibraryErrors** is 1 when one of the listed errors occurs, EGL sets the system variable **sysVar.errorCode** to a non-zero value. If the value of **VGVar.handleSysLibraryErrors** is 0 when one of the errors occurs, the program ends.

Of particular interest is the **sysVar.errorCode** value "00001000", which indicates that an exception was thrown by an invoked method or as a result of a class initialization.

When an exception is thrown, EGL stores it in the object space. If another exception occurs, the second exception takes the place of the first. You can use the identifier *caughtException* to access the last exception that occurred.

In an unusual situation, an invoked method throws not an exception but an error such as `OutOfMemoryError` or `StackOverflowError`. In such a case, the program ends regardless of the value of system variable `VGVar.handleSysLibraryErrors`.

The following Java code shows how a Java program can have multiple catch blocks to handle different kinds of exceptions. This code tries to create a `FileOutputStream` object. A failure causes the code to set an `errorType` variable and to store the exception that was thrown.

```
int errorType = 0;
Exception ex = null;

try
{
    java.io.FileOutputStream fOut =
        new java.io.FileOutputStream( "out.txt" );
}
catch ( java.io.IOException iox )
{
    errorType = 1;
    ex = iox;
}
catch ( java.lang.SecurityException sx )
{
    errorType = 2;
    ex = sx;
}
```

The following EGL code is equivalent to the previous Java code:

```
VGVar.handleSysLibraryErrors = 1;
errorType = 0;

JavaLib.storeNew( (objId)"fOut",
    "java.io.FileOutputStream", "out.txt" );

if ( sysVar.errorCode == "00001000" )
    exType = JavaLib.qualifiedTypeName( (objId)"caughtException" );

if ( exType == "java.io.IOException" )
    errorType = 1;
    JavaLib.storeCopy( (objId)"caughtException", (objId)"ex" );
else
    if ( exType == "java.lang.SecurityException" )
        errorType = 2;
        JavaLib.storeCopy( (objId)"caughtException", (objId)"ex" );
    end
end
end
```

#### Related reference

- "EGL library JavaLib" on page 934
- "Exception handling" on page 94
- "Primitive types" on page 34
- "getField()" on page 942
- "isNull()" on page 946
- "isObjID()" on page 947
- "qualifiedTypeName()" on page 948
- "remove()" on page 949
- "removeAll()" on page 950
- "setField()" on page 951
- "store()" on page 952



"storeCopy()" on page 954

"storeField()" on page 955

"storeNew()" on page 957

## getField()

The system function **JavaLib.getField** returns the value of a specified field of a specified object or class. **JavaLib.getField** is one of several Java access functions.

```
JavaLib.getField(  
  identifierOrClass javeObjIdOrClass in,  
  field STRING in)  
returns (result anyJavaPrimitive)
```

### *result*

The result field is required and receives the value of the field specified in the second argument. The following cases apply:

- If the received value is a BigDecimal, BigInteger, byte, short, int, long, float, or double, the result field must be a numeric data type. The characteristics do not need to match the value; for example, a float may be stored in a return variable that is declared with no decimal digits. For details on handling overflow, see *VGVar.handleOverflow* and *sysVar.overflowIndicator*.
- If the received value is a boolean, the result field must be of a numeric primitive type. The value is 1 for true, 0 for false.
- If the received value is a byte array, the result field must be of type HEX. For details on mismatched lengths, see *Assignments*.
- If the received value is a String or char, the result field must be of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE--
  - If the result field is of type MBCHAR, STRING, or UNICODE, the received value is always appropriate
  - If the result field is of type CHAR, problems can arise if the received value includes characters that correspond to DBCHAR characters
  - If the result field is of type DBCHAR, problems can arise if the received value includes Unicode characters that correspond to single-byte characters

For details on mismatched lengths, see *Assignments*.

- If the native Java method does not return a value or returns a null, error 00001004 occurs, as listed later.

### *identifierOrClass*

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. If you intend to specify a static field in the next argument, it is recommended that you specify a class in this argument.

EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

### *field*

The name of the field to read.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

An example is as follows:

```
myVar = JavaLib.getField( (objId)"myID", "myField" );
```

An error during processing of **JavaLib.getField** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001004	The method returned null, the method does not return a value, or the value of a field was null
00001005	The returned value does not match the type of the return variable
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

## Related concepts

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

### Related reference

"Assignments" on page 456

"BIN and the integer types" on page 50

"EGL library JavaLib" on page 934

"Exception handling" on page 94

"invoke()" on page 944

"isNull()" on page 946

"isObjID()" on page 947

"qualifiedTypeName()" on page 948

"remove()" on page 949

"removeAll()" on page 950

"setField()" on page 951

"store()" on page 952

"storeCopy()" on page 954

"storeField()" on page 955

"storeNew()" on page 957

## invoke()

The system function **JavaLib.invoke** invokes a method on a native Java object or class and may return a value. **JavaLib.invoke** is one of several Java access functions.

```
JavaLib.invoke(  
  identifierOrClass javaObjIdOrClass in,  
  method STRING in  
  {, argument anyEglPrimitive in})  
returns (result anyJavaPrimitive)
```

*result*

The result field, if present, receives a value from the native Java method.

If the native Java method returns a value, the result field is optional.

The following cases apply:

- If the returned value is a BigDecimal, BigInteger, byte, short, int, long, float, or double, the result field must be a numeric data type. The characteristics do not need to match the value; for example, a float may be stored in a result field that is declared with no decimal digits. For details on handling overflow, see *VGVar.handleOverflow* and *SysVar.overflowIndicator*.
- If the returned value is a boolean, the result field must be of a numeric primitive type. The value is 1 for true, 0 for false.
- If the returned value is a byte array, the result field must be of type HEX. For details on mismatched lengths, see *Assignments*.
- If the returned value is a String or char, the result field must be of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE--
  - If the result field is of type MBCHAR, STRING, or UNICODE, the returned value is always appropriate
  - If the result field is of type CHAR, problems can arise if the returned value includes characters that correspond to DBCHAR characters
  - If the result field is of type DBCHAR, problems can arise if the returned value includes Unicode characters that correspond to single-byte characters

For details on mismatched lengths, see *Assignments*.

- If the native Java method does not return a value or returns a null, the following cases apply:
  - No error occurs in the absence of a result field
  - An error occurs at run time if a result field is present; the error is 00001004, as listed later

*identifierOrClass*

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or an variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

Your code cannot invoke a method on an object until you have created an identifier for the object. A later example illustrates this point with `java.lang.System.out`, which refers to a `PrintStream` object.

#### *method*

The name of the method to call.

This argument is either a string literal or an variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

#### *argument*

A value passed to the method.

A cast may be required, as specified in *Java access (system words)*.

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a method parameter that is declared as an int.

To avoid losing precision, use an EGL float variable for a Java double, and an EGL smallfloat variable for a Java float. Using one of the other EGL types will probably result in a value being rounded.

The memory area in the invoking program does not change regardless of what the method does.

In the following example, the cast (objId) is required except as noted:

```
// call the constructor of the Java Date class and
// assign the new object to the identifier "date".
JavaLib.storeNew( (objId)"date", "java.util.Date");

// call the toString method of the new Date object
// and assign the output (today's date) to the chaItem.
// In the absence of the cast (objId), "date"
// refers to a class rather than an object.
chaItem = JavaLib.invoke( (objId)"date", "toString" );

// assign the standard output stream of the
// Java System class to the identifier "systemOut".
JavaLib.storeField( (objId)"systemOut", "java.lang.System", "out" );

// call the println method of the output
// stream and print today's date.
JavaLib.invoke( (objID)"systemOut", "println", chaItem );

// The use of "java.lang.System.out" as the first
// argument in the previous line would not have been
// valid, as the argument must either be a
// an identifier already in the object space or a class
// name. The argument cannot refer to a static field.
```

An error during processing of **JavaLib.invoke** can set **SysVar.errorCode** to a value listed in the next table.

Value in SysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java

Value in SysVar.errorCode	Description
00001004	The method returned null, the method does not return a value, or the value of a field was null
00001005	The returned value does not match the type of the return variable
00001006	The class of an argument cast to null could not be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

## Related concepts

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

### Related reference

"Assignments" on page 456

"BIN and the integer types" on page 50

"EGL library JavaLib" on page 934

"Exception handling" on page 94

"getField()" on page 942

"isNull()"

"isObjID()" on page 947

"qualifiedTypeName()" on page 948

"remove()" on page 949

"removeAll()" on page 950

"setField()" on page 951

"store()" on page 952

"storeCopy()" on page 954

"storeField()" on page 955

"storeNew()" on page 957

"Primitive types" on page 34

"overflowIndicator" on page 1069

"handleOverflow" on page 1083

## isNull()

The system function **JavaLib.isNull** returns a value (1 for true, 0 for false) to indicate whether a specified identifier refers to a null object. **JavaLib.isNull** is one of several Java access functions.

```
JavaLib.isNull(identifier javaObjId in)
returns (result INT)
```

*result*

A numeric field that receives one of two values: 1 for true, 0 for false. Use of a non-numeric field causes an error at validation time.

*identifier*

An identifier that refers to an object in the object space.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objID. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
// test whether an object is null
// and process accordingly
isNull = JavaLib.isNull( (objId)valueId );

if( isNull == 1 )
    error = 12;
end
```

An error during processing of **JavaLib.isNull** can set **SysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001001	The specified identifier was not in the object space

### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"EGL library JavaLib" on page 934

"getField()" on page 942

"invoke()" on page 944

"isObjID()"

"qualifiedTypeName()" on page 948

"remove()" on page 949

"removeAll()" on page 950

"setField()" on page 951

"store()" on page 952

"storeCopy()" on page 954

"storeField()" on page 955

"storeNew()" on page 957

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

## isObjID()

The system function **JavaLib.isObjID** returns a value (1 for true, 0 for false) to indicate whether a specified identifier is in the object space. **JavaLib.isObjID** is one of several Java access functions.

**JavaLib.isObjID**(*identifier* javaObjId in)  
returns (*result* INT)

*result*

A numeric item that receives one of two values: 1 for true, 0 for false. Use of a non-numeric item causes an error at validation time.

*identifier*

An identifier that refers to an object in the object space.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
// test whether an object is non-existent
// and process accordingly
isPresent = JavaLib.isObjID( (objId)valueId );

if( isPresent == 0 )
    error = 27;
end
```

No runtime errors are associated with **JavaLib.isObjID**.

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884  
"Java access functions" on page 935

#### **Related tasks**

"Syntax diagram for EGL statements and commands" on page 884

#### **Related reference**

"EGL library JavaLib" on page 934  
"getField()" on page 942  
"invoke()" on page 944  
"isNull()" on page 946  
"qualifiedTypeName()" on page 949  
"remove()" on page 949  
"removeAll()" on page 950  
"setField()" on page 951  
"store()" on page 952  
"storeCopy()" on page 954  
"storeField()" on page 955  
"storeNew()" on page 957

### **qualifiedTypeName()**

The system function **JavaLib.qualifiedTypeName** returns the fully qualified name of the class of an object in the EGL Java object space. **JavaLib.qualifiedTypeName** is one of several Java access functions.

```
JavaLib.qualifiedTypeName(identifier javaObjId in)
returns (result STRING)
```

#### *result*

The result field is required and must be of type CHAR, MBCHAR, or UNICODE--

- If the result field is of type MBCHAR or UNICODE, the received value is always appropriate
- If the result field is of type CHAR, problems can arise if the received value includes characters that correspond to DBCHAR characters

For details on mismatched lengths, see *Assignments*.

#### *identifier*

An identifier that refers to an object in the object space.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objId, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
myItem = JavaLib.qualifiedTypeName( (objId)"myId" );
```

An error during processing of **JavaLib.qualifiedTypeName** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001001	The object was null, or the specified identifier was not in the object space

### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"EGL library JavaLib" on page 934

"getField()" on page 942

"invoke()" on page 944

"isNull()" on page 946

"isObjID()" on page 947

"qualifiedTypeName()" on page 948

"remove()"

"removeAll()" on page 950

"setField()" on page 951

"store()" on page 952

"storeCopy()" on page 954

"storeField()" on page 955

"storeNew()" on page 957

### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

## remove()

The system function **JavaLib.remove** removes the specified identifier from the EGL Java object space. The object related to the identifier is also removed, but only if the identifier is the only one that refers to the object. If another identifier refers to the object, the object remains in the object space and is accessible by way of that other identifier.

**JavaLib.remove** is one of several Java access functions.

```
JavaLib.remove(identifier javaObjId in)
```

*identifier*

The identifier that refers to an object. No error occurs if the identifier is not found.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objID, as shown in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:



```
JavaLib.remove( (objId)myStoredObject );
```

No runtime errors are associated with **JavaLib.remove**.

**Note:** By invoking the system functions **JavaLib.remove** and **JavaLib.removeAll**, your code allows the Java Virtual Machine to handle garbage collection in the EGL Java object space. If you do not invoke a system function to remove an object from the object space, the memory is not recovered during the run time of any program that has access to the object space.

#### **Related concepts**

#### **Related tasks**

“Syntax diagram for EGL statements and commands” on page 884

#### **Related reference**

“EGL library JavaLib” on page 934

“getField()” on page 942

“invoke()” on page 944

“isNull()” on page 946

“isObjID()” on page 947

“qualifiedTypeName()” on page 948

“removeAll()”

“setField()” on page 951

“store()” on page 952

“storeCopy()” on page 954

“storeField()” on page 955

“storeNew()” on page 957

### **removeAll()**

The system function **JavaLib.removeAll** removes all identifiers and objects from the EGL Java object space. **JavaLib.removeAll** is one of several Java access functions.

```
JavaLib.removeAll( )
```

No runtime errors are associated with **JavaLib.removeAll**.

**Note:** By invoking the system functions **JavaLib.remove** and **JavaLib.removeAll**, your code allows the Java Virtual Machine to handle garbage collection in the EGL Java object space. If you do not invoke a system function to remove an object from the object space, the memory is not recovered during the run time of any program that has access to the object space.

#### **Related concepts**

#### **Related tasks**

“Syntax diagram for EGL statements and commands” on page 884

#### **Related reference**

“EGL library JavaLib” on page 934

“getField()” on page 942

“invoke()” on page 944

“isNull()” on page 946

“isObjID()” on page 947

"qualifiedTypeName()" on page 948  
 "remove()" on page 949  
 "setField()" on page 952  
 "store()" on page 952  
 "storeCopy()" on page 954  
 "storeField()" on page 955  
 "storeNew()" on page 957

## setField()

The system function **JavaLib.setField** sets the value of a field in a native Java object or class. **JavaLib.setField** is one of several Java access functions.

```

JavaLib.setField(
  identifierOrClass javaObjId in,
  field STRING in,
  value anyEglPrimitive in)

```

*identifierOrClass*

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

*field*

The name of the field to change.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

*value*

The value itself.

A cast may be required, as specified in Java access (system words).

The Java type-conversion rules are in effect. No error occurs, for example, if you assign a short to a field that is declared as an int.

An example is as follows:

```

JavaLib.setField( (objID)"myId", "myField",
  (short)myNumItem );

```

An error during processing of **JavaLib.setField** can set **SysVar.errorCode** to a value listed in the next table.

Value in SysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded

Value in SysVar.errorCode	Description
00001003	The EGL primitive type does not match the type expected in Java
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

### Related concepts

“Syntax diagram for EGL statements and commands” on page 884

### Related reference

“EGL library JavaLib” on page 934

“getField()” on page 942  
 “invoke()” on page 944  
 “isNull()” on page 946  
 “isObjID()” on page 947  
 “qualifiedTypeName()” on page 948  
 “remove()” on page 949  
 “removeAll()” on page 950  
 “store()”  
 “storeCopy()” on page 954  
 “storeField()” on page 955  
 “storeNew()” on page 957

### store()

The system function **JavaLib.store** invokes a method and places the returned object (or null) into the EGL Java object space, along with a specified identifier. If the identifier is already in the object space, the action is equivalent to the following steps:

- Running **JavaLib.remove** on the identifier to remove the object that was related to that identifier
- Relating the **JavaLib.store**-returned object with the target identifier

If the method returns a Java primitive instead of an object, EGL stores an object that represents the primitive; for example, if the method returns an int, EGL stores an object of type java.lang.Integer.

**JavaLib.store** is one of several Java access functions.

```

JavaLib.store(
  storeId javaObjId in,
  identifierOrClass javaObjId in,
  method STRING in
  {, argument anyEglPrimitive in} )

```

*storeId*

The identifier to store with the returned object.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

#### *identifierOrClass*

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

#### *method*

The method to invoke.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

#### *argument*

A value passed to the method.

A cast may be required, as specified in Java access (system words).

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a method parameter that is declared as an int.

To avoid losing precision, use an EGL float item for a Java double, and an EGL smallfloat item for a Java float. Using one of the other EGL types will probably result in a value being rounded.

The memory area in the invoking program does not change regardless of what the method does.

An example is as follows:

```
JavaLib.store( (objId)"storeId", (objId)"myId",  
              "myMethod", 36 );
```

An error during processing of **JavaLib.store** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java

Value in sysVar.errorCode	Description
00001006	The class of an argument cast to null could not be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"EGL library JavaLib" on page 934

"getField()" on page 942

"invoke()" on page 944

"isNull()" on page 946

"isObjID()" on page 947

"qualifiedTypeName()" on page 948

"remove()" on page 949

"removeAll()" on page 950

"setField()" on page 951

"storeCopy()"

"storeField()" on page 955

"storeNew()" on page 957

## storeCopy()

The system function **JavaLib.storeCopy** creates a new identifier based on another in the object space, so that both refer to the same object. If the source identifier is not in the object space, a null is stored for the target identifier and no error occurs. If the target identifier is already in the object space, the action is equivalent to the following steps:

- Running **JavaLib.remove** on the target identifier to remove the object that was related to that identifier
- Relating the source object with the target identifier

**JavaLib.storeCopy** is one of several Java access functions.

```
JavaLib.storeCopy(
    sourceId javaObjId in,
    targetId javaObjId in)
```

*sourceId*

An identifier that refers to an object in the object space or to null.

This argument is either a string literal or a variable of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objId, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

*targetId*

The new identifier, which refers to the same object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, STRING, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

An example is as follows:

```
JavaLib.storeCopy( (objId)"sourceId", (objId)"targetId" );
```

No runtime errors are associated with **JavaLib.storeCopy**.

### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"EGL library JavaLib" on page 934

"getField()" on page 942

"invoke()" on page 944

"isNull()" on page 946

"isObjID()" on page 947

"qualifiedTypeName()" on page 948

"remove()" on page 949

"removeAll()" on page 950

"setField()" on page 951

"store()" on page 952

"storeField()"

"storeNew()" on page 957

## storeField()

The system function **JavaLib.storeField** places the value of a class field or object field into the EGL Java object space. If the identifier used to store the object is already in the object space, the action is equivalent to the following steps:

- Running **JavaLib.remove** on the identifier to remove the object that was related to the identifier
- Relating the new object with the identifier

If the class or object field contains a Java primitive instead of an object, EGL stores an object that represents the primitive; for example, if the field contains an int, EGL stores an object of type java.lang.Integer.

```
JavaLib.storeField(  
  storeId javaObjId in,  
  identifierOrClass javaObjIdOrClass in,  
  field STRING in)
```

*storeId*

The identifier to store with the object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

*identifierOrClass*

This argument is one of the following entities:

- An identifier that refers to an object in the object space; or
- The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. If you are specifying an identifier of an object, the identifier must be cast to objID, as in a later example. If you intend to specify a static field in the next argument, it is recommended that you specify a class in this argument.

EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

#### *field*

The name of the field that refers to an object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. Single- and double-byte blanks are stripped from the beginning and end of the string, which is case sensitive.

An example is as follows:

```
JavaLib.storeField( (objId)"myStoreId",
  (objId)"myId", "myField");
```

An error during processing of **JavaLib.storeField** can set **sysVar.errorCode** to a value listed in the next table.

Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001009	An identifier rather than a class name must be specified; the method or field is not static

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

#### **Related reference**

"EGL library JavaLib" on page 934

"getField()" on page 942

"invoke()" on page 944

"isNull()" on page 946

"isObjID()" on page 947

"qualifiedTypeName()" on page 948

"remove()" on page 949

"removeAll()" on page 950

"setField()" on page 951

"store()" on page 952  
"storeCopy()" on page 954  
"storeNew()"

#### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

### storeNew()

The system function **JavaLib.storeNew** invokes the constructor of a class and places the new object into the EGL Java object space. If the identifier is already in the object space, the action is equivalent to the following steps:

- Running **JavaLib.remove** on the identifier to remove the object previously associated with the identifier
- Relating the new object with the identifier

**JavaLib.storeNew** is one of several Java access functions.

```
JavaLib.storeNew(  
  storeId javaObjId in,  
  class STRING in  
  {, argument anyEglPrimitive in})
```

#### *storeId*

The identifier to store with the new object.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. The identifier must be cast to objID, as in a later example. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

#### *class*

The fully qualified name of a Java class.

This argument is either a string literal or an item of type CHAR, DBCHAR, MBCHAR, or UNICODE. EGL strips single- and double-byte blanks from the beginning and end of the argument value, which is case sensitive.

#### *argument*

A value passed to the constructor.

A cast may be required, as specified in *Java access (system words)*.

The Java type-conversion rules are in effect. No error occurs, for example, if you pass a short to a constructor parameter that is declared as an int.

To avoid losing precision, use an EGL float item for a Java double, and an EGL smallfloat item for a Java float. Using one of the other EGL types will probably result in a value being rounded.

The memory area in the invoking program does not change regardless of what the constructor does.

An example is as follows:

```
JavaLib.storeNew( (objId)"storeId", "myClass", 36 );
```

An error during processing of **JavaLib.storeNew** can set **sysVar.errorCode** to a value listed in the next table.



Value in sysVar.errorCode	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization
00001001	The object was null, or the specified identifier was not in the object space
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded
00001003	The EGL primitive type does not match the type expected in Java
00001006	The class of an argument cast to null could not be loaded
00001007	A SecurityException or IllegalAccessException was thrown during an attempt to get information about a method or field; or an attempt was made to set the value of a field that was declared final
00001008	The constructor cannot be called; the class name refers to an interface or abstract class

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"EGL library JavaLib" on page 934

"getField()" on page 942

"invoke()" on page 944

"isNull()" on page 946

"isObjID()" on page 947

"qualifiedTypeName()" on page 948

"remove()" on page 949

"removeAll()" on page 950

"setField()" on page 951

"store()" on page 952

"storeCopy()" on page 954

"storeField()" on page 955

#### Related tasks

"Syntax diagram for EGL statements and commands" on page 884

## EGL library LobLib

The next table lists the functions in the library LobLib.

System function/Invocation	Description
attachBlobToFile( <i>blobVariable</i> , <i>fileName</i> )	Copies the data referenced by a variable of type BLOB into a specified file.
attachBlobToTempFile( <i>blobVariable</i> )	Copies the data referenced by a variable of type BLOB into a unique, temporary system file.

System function/Invocation	Description
<code>attachClobToFile(<i>clobVariable</i>, <i>fileName</i>)</code>	Copies the data referenced by a variable of type CLOB into a specified file.
<code>attachClobToTempFile(<i>clobVariable</i> )</code>	Copies the data referenced by a variable of type CLOB into a unique, temporary system file.
<code>freeBlob(<i>blobVariable</i>)</code>	Releases the resources used by a variable of type BLOB.
<code>freeClob(<i>clobVariable</i>)</code>	Releases the resources used by a variable of type CLOB.
<code>result = getBlobLen(<i>blobVariable</i> )</code>	Returns the number of bytes in the value referenced by a variable of type BLOB.
<code>result = getClobLen(<i>clobVariable</i>)</code>	Returns the number of characters referenced by a variable of type CLOB.
<code>result = getStrFromClob(<i>clobVariable</i>)</code>	Returns a string that corresponds to the value referenced by a variable of type CLOB.
<code>result = getSubStrFromClob(<i>clobVariable</i>, <i>pos</i>, <i>length</i>)</code>	Returns a substring from the value referenced by a variable of type CLOB.
<code>loadBlobFromFile(<i>blobVariable</i>, <i>fileName</i>)</code>	Copies the data from a specified file to a memory area referenced by a variable of type BLOB.
<code>loadClobFromFile(<i>blobVariable</i>, <i>fileName</i>)</code>	Copies the data from a specified file to a memory area referenced by a variable of type CLOB.
<code>setClobFromString(<i>clobVariable</i>, <i>str</i>)</code>	Copies a string into a memory area referenced by a variable of type CLOB.
<code>setClobFromStringAtPosition(<i>clobVariable</i>, <i>pos</i>, <i>str</i>)</code>	Copies a string into a memory area referenced by a variable of type CLOB, starting at a specified position in the memory area.
<code>truncateBlob(<i>blobVariable</i>, <i>length</i>)</code>	Truncates the value referenced by a variable of type BLOB.
<code>truncateClob(<i>clobVariable</i>, <i>length</i>)</code>	Truncates the value referenced by a variable of type CLOB.
<code>updateBlobToFile(<i>blobVariable</i>, <i>fileName</i>)</code>	Copies the data referenced by a variable of type BLOB into a specified file.
<code>updateClobToFile(<i>blobVariable</i>, <i>fileName</i>)</code>	Copies the data referenced by a variable of type CLOB into a specified file.

## attachBlobToFile()

The system function **LobLib.attachBlobToFile** copies the data referenced by a variable of type BLOB into a specified file.

```
LobLib.attachBlobToFile(
    blobVariable BLOB inOut,
    fileName STRING in)
```

*blobVariable*

The variable of type BLOB.

*fileName*

The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“BLOB” on page 49

“EGL library LobLib” on page 958

## **attachBlobToTempFile()**

The system function **LobLib.attachBlobToTempFile** copies the data referenced by a variable of type BLOB into a unique, temporary system file. This function minimizes the memory used at run time.

**LobLib.attachBlobToTempFile**(*blobVariable* **BLOB** in)

*blobVariable*

The variable of type BLOB.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“BLOB” on page 49

“EGL library LobLib” on page 958

## **attachClobToFile()**

The system function **LobLib.attachClobToFile** copies the data referenced by a variable of type CLOB into a specified file.

**LobLib.attachClobToFile**(  
    *clobVariable* **CLOB** inOut,  
    *fileName* **STRING** in)

*clobVariable*

The variable of type CLOB.

*fileName*

The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“CLOB” on page 48

“EGL library LobLib” on page 958

## **attachClobToTempFile()**

The system function **LobLib.attachClobToTempFile** copies the data referenced by a variable of type CLOB into a unique, temporary system file. This function minimizes the memory used at run time.

**LobLib.attachClobToTempFile**(*clobVariable* **CLOB** in)

*clobVariable*

The variable of type CLOB.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“CLOB” on page 48

“EGL library LobLib” on page 958

**freeBlob()**

The system function **LobLib.freeBlob** releases any resources used by a variable of type BLOB.

**LobLib.freeBlob**(*blobVariable* **BLOB** inOut)

*blobVariable*

The variable of type BLOB.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“BLOB” on page 49

“EGL library LobLib” on page 958

**freeClob()**

The system function **LobLib.freeClob** releases the resources used by a variable of type CLOB.

**LobLib.freeClob**(*clobVariable* **CLOB** inOut)

*clobVariable*

The variable of type CLOB.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“CLOB” on page 48

“EGL library LobLib” on page 958

**getBlobLen()**

The system function **LobLib.getBlobLen** returns the number of bytes in the value referenced by a variable of type BLOB.

**LobLib.getBlobLen**(*blobVariable* **BLOB** in)  
returns (*result* **BIGINT**)

*result*

The number of bytes.

*blobVariable*

The variable of type BLOB.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“BLOB” on page 49

“EGL library LobLib” on page 958

## getClobLen()

The system function **LobLib.getClobLen** returns the number of characters referenced by a variable of type CLOB.

```
LobLib.getClobLen(clobVariable CLOB in)  
returns (result BIGINT)
```

*result*

The number of characters.

*clobVariable*

The variable of type CLOB.

### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"CLOB" on page 48

"EGL library LobLib" on page 958

## getStrFromClob()

The system function **LobLib.getStrFromClob** returns a string that corresponds to the value referenced by a variable of type CLOB.

```
LobLib.getStrFromClob(clobVariable CLOB in)  
returns (result STRING)
```

*result*

The returned string.

*clobVariable*

The variable of type CLOB.

### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"CLOB" on page 48

"EGL library LobLib" on page 958

## getSubStrFromClob()

The system function **LobLib.getSubStrFromClob** returns a substring from the value referenced by a variable of type CLOB.

```
LobLib.getSubStrFromClob(  
  clobVariable CLOB in,  
  pos BIGINT in,  
  length BIGINT in)  
returns (result STRING)"
```

*result*

A value of type STRING.

*clobVariable*

The variable of type CLOB.

*pos*

Identifies the numeric position of the character that starts the substring. The first character in the CLOB variable is at position 1.

*length*

Identifies the number of characters in the substring.

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"CLOB" on page 48

"EGL library LobLib" on page 958

**loadBlobFromFile()**

The system function **LobLib.loadBlobFromFile** copies the data from a specified file to a memory area referenced by a variable of type BLOB.

```
LobLib.loadBlobFromFile(  
    blobVariable BLOB inOut,  
    fileName STRING in)
```

*blobVariable*

The variable of type BLOB.

*fileName*

The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"BLOB" on page 49

"EGL library LobLib" on page 958

**loadClobFromFile()**

The system function **LobLib.loadClobFromFile** copies the data from a specified file to a memory area referenced by a variable of type CLOB.

```
LobLib.loadClobFromFile(  
    clobVariable CLOB inOut,  
    fileName STRING in)
```

*clobVariable*

The variable of type CLOB.

*fileName*

The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"CLOB" on page 48

"EGL library LobLib" on page 958

**setClobFromString()**

The system function **LobLib.setClobFromString** copies a string into a memory area referenced by a variable of type CLOB.

```
LobLib.setClobFromString(
  clobVariable CLOB inOut,
  str STRING in)
```

*clobVariable*

The variable of type CLOB.

*str* The string to be copied.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"CLOB" on page 48

"EGL library LobLib" on page 958

### setClobFromStringAtPosition()

The system function **LobLib.setClobFromStringAtPosition** copies a string into the memory area referenced by a variable of type CLOB, starting at a specified position in the memory area.

```
LobLib.setClobFromStringAtPosition(
  clobVariable CLOB inOut,
  pos BIGINT in
  str STRING in)
```

*clobVariable*

The variable of type CLOB.

*pos*

The character position in the value referenced by *clobVariable*. The first character in the CLOB variable is at position 1.

*str* The string to be copied.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"CLOB" on page 48

"EGL library LobLib" on page 958

### truncateBlob()

The system function **LobLib.truncateBlob** truncates the value referenced by a variable of type BLOB.

```
LobLib.truncateBlob(
  blobVariable BLOB inOut,
  length BIGINT in)
```

*blobVariable*

A variable of type BLOB.

*length*

The number of bytes in the output.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

“BLOB” on page 49

“EGL library LobLib” on page 958

### truncateClob()

The system function **LobLib.truncateClob** truncates the value referenced by a variable of type CLOB.

```
LobLib.truncateClob(  
    clobVariable CLOB inOut,  
    length BIGINT in)
```

*clobVariable*

A variable of type CLOB.

*length*

The number of bytes (not characters) in the output.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“CLOB” on page 48

“EGL library LobLib” on page 958

### updateBlobToFile()

The system function **LobLib.updateBlobToFile** copies the data referenced by a variable of type BLOB into a specified file. If the file exists, the function first erases the content of the file; otherwise, the function creates the file.

```
LobLib.updateBlobToFile(  
    blobVariable BLOB inOut,  
    fileName STRING in)
```

*blobVariable*

The variable of type BLOB.

*fileName*

The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“BLOB” on page 49

“EGL library LobLib” on page 958

### updateClobToFile()

The system function **LobLib.updateClobToFile** copies the data referenced by a variable of type CLOB into a specified file. If the file exists, the function first erases the content of the file; otherwise, the function creates the file.

```
LobLib.updateClobToFile(  
    clobVariable CLOB inOut,  
    fileName STRING in)
```

*clobVariable*

The variable of type CLOB.



*fileName*

The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“CLOB” on page 48

“EGL library LobLib” on page 958

## EGL library MathLib

The next table lists the functions in the system library MathLib.

**Note:** The field *numericField* is of type BIGINT, BIN, DECIMAL, HEX, INT, NUM, NUMC, PACF, SMALLINT, FLOAT, or SMALLFLOAT.

A field of type HEX (length 8) is assumed to be a single-precision, 4-byte floating-point number that is native to the runtime environment; and a field of type HEX (length 16) is assumed to be a double-precision, 8-byte floating-point number that is native to the runtime environment.

System function/Invocation	Description
<i>result</i> = abs ( <i>numericField</i> )	Returns absolute value of <i>numericField</i>
<i>result</i> = acos ( <i>numericField</i> )	Returns arccosine of <i>numericField</i>
<i>result</i> = asin ( <i>numericField</i> )	Returns arcsine of <i>numericField</i>
<i>result</i> = atan ( <i>numericField</i> )	Returns arctangent of <i>numericField</i>
<i>result</i> = atan2 ( <i>numericField1</i> , <i>numericField2</i> )	Computes the principal value of the arc tangent of <i>numericField1</i> / <i>numericField2</i> , using the signs of both arguments to determine the quadrant of the return value
<i>result</i> = ceiling ( <i>numericField</i> )	Returns smallest integer not less than <i>numericField</i>
<i>result</i> = compareNum ( <i>numericField1</i> , <i>numericField2</i> )	Returns a result (-1, 0, or 1) that indicates whether <i>numericField1</i> is less than, equal to, or greater than <i>numericField2</i>
<i>result</i> = cos ( <i>numericField</i> )	Returns cosine of <i>numericField</i>
<i>result</i> = cosh ( <i>numericField</i> )	Returns hyperbolic cosine of <i>numericField</i>
<i>result</i> = exp ( <i>numericField</i> )	Returns exponential value of <i>numericField</i>
<i>result</i> = floatingAssign ( <i>numericField</i> )	Returns <i>numericField</i> as a double-precision floating-point number
<i>result</i> = floatingDifference ( <i>numericField1</i> , <i>numericField2</i> )	Returns the difference between <i>numericField1</i> and <i>numericField2</i>
<i>result</i> = floatingMod ( <i>numericField1</i> , <i>numericField2</i> )	Calculates the floating point remainder of <i>numericField1</i> divided by <i>numericField2</i> , with the result having the same sign as <i>numericField1</i>
<i>result</i> = floatingProduct ( <i>numericField1</i> , <i>numericField2</i> )	Returns product of <i>numericField1</i> and <i>numericField2</i>
<i>result</i> = floatingQuotient ( <i>numericField1</i> , <i>numericField2</i> )	Returns quotient of <i>numericField1</i> divided by <i>numericField2</i>

System function/Invocation	Description
<i>result</i> = floatingSum ( <i>numericField1</i> , <i>numericField2</i> )	Returns sum of <i>numericField1</i> and <i>numericField2</i>
<i>result</i> = floor ( <i>numericField</i> )	Returns the largest integer not greater than <i>numericField</i>
<i>result</i> = frexp ( <i>numericField</i> , <i>integer</i> )	Splits a number into a normalized fraction in the range of .5 to 1 (which is the returned value) and a power of 2 (which is returned in <i>integer</i> )
<i>result</i> = ldexp ( <i>numericField</i> , <i>integer</i> )	Returns <i>numericField</i> multiplied by 2 to the power of <i>integer</i>
<i>result</i> = log ( <i>numericField</i> )	Returns the natural logarithm of <i>numericField</i>
<i>result</i> = log10 ( <i>numericField</i> )	Returns the base 10 logarithm of <i>numericField</i>
<i>result</i> = maximum ( <i>numericField1</i> , <i>numericField2</i> )	Returns the greater of <i>numericField1</i> and <i>numericField2</i>
<i>result</i> = minimum ( <i>numericField1</i> , <i>numericField2</i> )	Returns the lesser of <i>numericField1</i> and <i>numericField2</i>
<i>result</i> = modf ( <i>numericField1</i> , <i>numericField2</i> )	Splits <i>numericField1</i> into integral and fractional parts, both with the same sign as <i>numericField1</i> ; places the integral part in <i>numericField2</i> ; and returns the fractional part
<i>result</i> = pow ( <i>numericField1</i> , <i>numericField2</i> )	Returns <i>numericField1</i> raised to the power of <i>numericField2</i>
<i>result</i> = precision ( <i>numericField</i> )	Returns the maximum precision (in decimal digits) for <i>numericField</i>
<i>result</i> = round ( <i>numericField</i> [, <i>integer</i> ]) <i>result</i> = mathLib.round( <i>numericExpression</i> )	Rounds a number or expression to a nearest value (for example, to the nearest thousands) and returns the result
<i>result</i> = sin ( <i>numericField</i> )	Returns sine of <i>numericField</i>
<i>result</i> = sinh ( <i>numericField</i> )	Returns hyperbolic sine of <i>numericField</i>
<i>result</i> = sqrt ( <i>numericField</i> )	Returns the square root of <i>numericField</i> if <i>numericField</i> is greater than or equal to zero
<i>result</i> = stringAsDecimal ( <i>numberAsText</i> )	Accepts a character value (like "98.6") and returns the equivalent value of type DECIMAL
<i>result</i> = stringAsFloat ( <i>numberAsText</i> )	Accepts a character value (like "98.6") and returns the equivalent value of type FLOAT
<i>result</i> = stringAsInt ( <i>numberAsText</i> )	Accepts a character value (like "98") and returns the equivalent value of type BIGINT
<i>result</i> = tan ( <i>numericField</i> )	Returns the tangent of <i>numericField</i>
<i>result</i> = tanh ( <i>numericField</i> )	Returns the hyperbolic tangent of <i>numericField</i>

## abs()

The system function **MathLib.abs** returns the absolute value of a number.

**MathLib.abs**(*numericField* **mathLibNumber** in)  
returns (*result* **mathLibTypeDependentResult** t)

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The absolute value of *numericItem* is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric item or HEX item, as described in *Mathematical (system words)*.

**MathLib.abs** works on every target system. In relation to Java programs, EGL uses one of the `abs()` methods in the Java `StrictMath` class so that the runtime behavior is the same for every Java Virtual Machine.

**Example:**

```
myItem = -5;  
result = MathLib.abs(myItem); // result = 5
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**acos()**

The system function **MathLib.acos** returns the arccosine of an argument, in radians.

```
MathLib.acos(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The returned value (between 0.0 and pi) is in radians and is converted to the format of *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before the calculation occurs. If the value is not between -1 and 1, an error occurs.

**MathLib.acos** works on every target system. In relation to Java programs, EGL uses the `acos()` method in the Java `StrictMath` class so that the runtime behavior is the same for every Java Virtual Machine.

**Example:**

```
result = MathLib.acos(myItem);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**asin()**

The system function **MathLib.asin** returns the arcsine of a number that is in the range of -1 to 1. The result is in radians and is in the range of -pi/2 to pi/2.

```
MathLib.asin(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the `MathLib.asin` function is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before the `mathLib.asin` function is called.

**Example:**

```
result = MathLib.asin(myItem);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

## **atan()**

The system function **MathLib.atan** returns the arctangent of a number. The result is in radians and is in the range of  $-\pi/2$  and  $\pi/2$ .

```
MathLib.atan(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **MathLib.atan** is converted to the format of *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **MathLib.atan** is called.

**Example:**

```
result = MathLib.atan(myItem);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

## **atan2()**

The system function **MathLib.atan2** computes the principal value of the arc tangent of  $y/x$ , using the signs of both arguments to determine the quadrant of the return value. The result is in radians and is in the range of  $-\pi$  to  $\pi$ .

```
MathLib.atan2(  
  numericField1 mathLibNumber in,  
  numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **MathLib.atan2** is converted to the format of *result* and returned in *result*.

*numericField1*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **MathLib.atan2** is called. *numericField1* is the y value.

*numericField2*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating point before **MathLib.atan2** is called. *numericField2* is the x value.

**Example:**

```
myItemY = 1;
myItemX = 5;

// returns pi/2
result = MathLib.atan2(myItemY, myItemX);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**ceiling()**

The system function **MathLib.ceiling** returns the smallest integer not less than a specified number.

```
MathLib.ceiling(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The smallest integer not less than *numericItem* is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*.

**Example:**

```
myItem = 4.5;
result = MathLib.ceiling(myItem); // result = 5
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**compareNum()**

The system function **MathLib.compareNum** returns a result (-1, 0, or 1) that indicates whether the first of two numbers is less than, equal to, or greater than the second.

```
MathLib.compareNum(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

*result*

Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. This item receives one of the following values:

- 1      *numericField1* is less than *numericField2*.
- 0        *numericField1* is equal to *numericField2*.
- 1        *numericField1* is greater than *numericField2*.

*numericField1*

Any numeric or HEX item, as described in *Mathematical (system words)*.

*numericField2*

Any numeric or HEX item, as described in *Mathematical (system words)*.

#### Example:

```
myItem01 = 4
myItem02 = 7

result = MathLib.compareNum(myItem01,myItem02);

// result = -1
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library MathLib” on page 966

### cos()

The system function **MathLib.cos** returns the cosine of a number. The returned value is in the range of -1 to 1.

```
MathLib.cos(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **MathLib.cos** is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before **MathLib.cos** is called.

#### Example:

```
result = MathLib.cos(myItem);
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library MathLib” on page 966

### cosh()

The system function **MathLib.cosh** returns the hyperbolic cosine of a number.

```
MathLib.cosh(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by **mathLib.cosh** is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before `mathLib.cosh` is called.

**Example:**

```
result = MathLib.cosh(myItem);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**exp()**

The system function **MathLib.exp** returns *e* raised to the power of a number.

```
MathLib.exp(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *MathLib*. The value returned by **MathLib.exp** is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *MathLib*. The item is converted to double-precision floating-point before **MathLib.exp** is called.

**Example:**

```
result = MathLib.exp(myItem);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**floatingAssign()**

The system function **MathLib.floatingAssign** returns *numericItem* as a double-precision floating-point number. The function assigns the value of BIN, DECIMAL, NUM, NUMC, or PACKF items to floating-point numbers that are defined as HEX items, and vice versa.

```
MathLib.floatingAssign(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point number is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before being assigned to the result.

**Example:**

```
result = MathLib.floatingAssign(myItem);
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library MathLib” on page 966

## floatingDifference()

The system function **MathLib.floatingDifference** subtracts the second of two numbers from the first and returns the difference. The function is implemented using double-precision floating-point arithmetic.

```
MathLib.floatingDifference(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

### result

Any numeric or HEX item, as described in *Mathematical (system words)*. The difference is converted to the format of *result* and returned in *result*.

### numericField1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the difference is calculated.

### numericField2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the difference is calculated.

### Example:

```
result = MathLib.floatingDifference(myItem01,myItem02);
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library MathLib” on page 966

## floatingMod()

The system function **MathLib.floatingMod** returns the floating-point remainder of one number divided by another. The result has the same sign as the numerator. A domain exception is raised if the denominator equals zero.

```
MathLib.floatingMod(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

### result

Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point remainder is converted to the format of *result* and returned in *result*.

### numericField1

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

### numericField2

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.



**Example:**

```
result = MathLib.floatingMod(myItem01,myItem02);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**floatingProduct()**

The system function **MathLib.floatingProduct** returns the product of two numbers. The function is implemented using double-precision floating-point arithmetic.

```
MathLib.floatingProduct(
    numericField1 mathLibNumber in,
    numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

**result**

Any numeric or HEX item, as described in *Mathematical (system words)*. The product is converted to the format of *result* and returned in *result*.

**numericField1**

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

**numericField2**

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

**Example:**

```
result = MathLib.floatingProduct(myItem01,myItem02);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**floatingQuotient()**

The system function **MathLib.floatingQuotient** returns the quotient of one number divided by another. A domain exception is raised if the denominator equals zero. The function is implemented using double-precision floating-point arithmetic.

```
MathLib.floatingQuotient(
    numericField1 mathLibNumber in,
    numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

**result**

Any numeric or HEX item, as described in *Mathematical (system words)*. The quotient is converted to the format of *result* and returned in *result*.

**numericField1**

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the quotient is calculated.

**numericField2**

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the quotient is calculated.

**Example:**

```
result = MathLib.floatingQuotient(myItem01,myItem02);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**floatingSum()**

The system function **MathLib.floatingSum** returns the sum of two numbers. The function is implemented using double-precision floating-point arithmetic.

```
MathLib.floatingSum(
    numericField1 mathLibNumber in,
    numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The sum is converted to the format of *result* and returned in *result*.

*numericField1*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the sum is calculated.

*numericField2*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before the sum is calculated.

**Example:**

```
result = MathLib.floatingSum(myItem01,myItem02);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**floor()**

The system function **MathLib.floor** returns the largest integer not greater than a specified number.

```
MathLib.floor(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The largest integer not greater than *numericField* is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*.

**Example:**

```
myItem = 4.6;
result = MathLib.floor(myItem); // result = 4
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library MathLib” on page 966

## frexp()

The system function **MathLib.frexp** splits a number into a normalized fraction in the range of .5 to 1 (which is returned as the *result*) and a power of 2 (which is returned in *exponent*).

```
MathLib.frexp(  
    numericField mathLibNumber in,  
    exponent mathLibInteger inOut)  
returns (result mathLibTypeDependentResult)
```

### result

Any numeric or HEX item, as described in *Mathematical (system words)*. The floating-point fraction is converted to the format of *result* and returned in *result*.

### numericField

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

### exponent

Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

### Example:

```
result = MathLib.frexp(myItem,myInteger);
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library MathLib” on page 966

## Ldexp()

The system function **MathLib.Ldexp** returns the value of a specified number that is multiplied by the following value: two to the power of *exponent*.

```
MathLib.Ldexp(  
    numericField mathLibNumber in,  
    exponent mathLibInteger in)  
returns (result mathLibTypeDependentResult)
```

### result

Any numeric or HEX item, as described in *Mathematical (system words)*. The calculated value is converted to the format of *result* and returned in *result*.

### numericField

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

### exponent

Item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

### Example:

```
result = MathLib.Ldexp(myItem,myInteger);
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library MathLib” on page 966

### log()

The system function **MathLib.log** returns the natural logarithm of a number.

```
MathLib.log(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

#### *result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **mathLib.log** function is converted to the format of *result* and returned in *result*.

#### *numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

#### Example:

```
result = MathLib.log(myItem);
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library MathLib” on page 966

### log10()

The system function **MathLib.log10** returns the base 10 logarithm of a number.

```
MathLib.log10(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

#### *result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **log10** function is converted to the format of *result* and returned in *result*.

#### *numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

#### Example:

```
result = MathLib.log10(myItem);
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library MathLib” on page 966

### maximum()

The system function **MathLib.maximum** returns the greater of two numbers.

```
MathLib.maximum(  
  numericField1 mathLibNumber in,  
  numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The greater of two numbers is converted to the format of *result* and returned in *result*.

*numericField1*

Any numeric or HEX item, as described in *Mathematical (system words)*.

*numericField2*

Any numeric or HEX item, as described in *Mathematical (system words)*.

**Example:**

```
result = MathLib.maximum(myItem01,myItem02);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**minimum()**

The system function **MathLib.minimum** returns the lesser of two numbers.

```
MathLib.minimum(  
  numericField1 mathLibNumber in,  
  numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The lesser of two numbers is converted to the format of *result* and returned in *result*.

*numericField1*

Any numeric or HEX item, as described in *Mathematical (system words)*.

*numericField2*

Any numeric or HEX item, as described in *Mathematical (system words)*.

**Example:**

```
result = MathLib.minimum(myItem01,myItem02);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**modf()**

The system function **MathLib.modf** splits a number into integral and fractional parts, both with the same sign as the number. The fractional part is returned in *result* and the integral part is returned in *numericField2*.

```
MathLib.modf(  
  numericField1 mathLibNumber in,  
  numericField2 mathLibNumber inOut)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The fractional part of *numericField1* is converted to the format of *result* and returned in *result*.

*numericField1*

Any numeric or HEX item, as described in *Mathematical (system words)*.

*numericField2*

Any numeric or HEX item, as described in *Mathematical (system words)*. The integral part of *numericField1* is converted to the format of *numericField2* and returned in *numericField2*.

**Example:**

```
result = MathLib.modf(myItem01,myItem02);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**pow()**

The system function **MathLib.pow** returns a number raised to the power of a second number. A domain exception is raised if on pow(x,y) the value of x is negative and y is non-integral, or the value of x is 0.0 and y is negative.

```
MathLib.pow(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The result of the mathLib.pow function is converted to the format of *result* and returned in *result*.

*numericField1*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

*numericField2*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

**Example:**

```
result = MathLib.pow(myItem01,myItem02);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**precision()**

The system function **MathLib.precision** returns the maximum precision (in decimal digits) for a number. For floating-point numbers (8-digit HEX for standard-precision floating-point number or 16-digit HEX for double-precision floating-point number), the precision is the maximum number of decimal digits that can be represented in the number for the system on which the program is running.

```
MathLib.precision(numericField mathLibNumber in)  
returns (result INT)
```

*result*

An item that receives the precision of *numericItem*. The *result* item is defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*.

**Example:**

```
result = MathLib.precision(myItem);
```

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"EGL library MathLib" on page 966

**round()**

The system function **MathLib.round** rounds a number or expression to a nearest value (for example, to the nearest thousands) and returns the result.

```
MathLib.round(  
    numericField mathLibNumber in  
    [, powerOf10 mathLibInteger in  
    ]  
)  
returns (result mathLibTypeDependentResult)  
MathLib.round(numericExpression anyNumericExpression in  
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value produced by the rounding operation is converted to the format of *result* and returned in *result*.

The maximum supported length in this case is 31 rather than 32 because rounding occurs as follows:

- Add five to the digit in *result* at a precision one higher than the precision of the result digit
- Truncate the result

A numeric overflow occurs at run time if more than 31 digits are used in the calculation and if EGL cannot determine the violation at development time.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*.

*numericExpression*

A numeric expression other than simply a numeric item. If you specify an operator, you cannot specify a value for *powerOf10*.

You cannot use **MathLib.round** with the remainder operator (%).

*powerOf10*

An integer that determines the value to which the number is rounded:

- If the integer is positive, the number is rounded to a nearest value equal to 10 to the power of *powerOf10*. If integer is 3, for example, the number is rounded to the nearest thousands.
- The same is true if the integer is zero or negative; in that case, the number is rounded to the specified number of decimal places.

If you do not specify *powerOf10*, **MathLib.round** rounds to the number of decimal places in *result*.

The integer is defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

**Examples:** In the next example, item balance is rounded to the nearest thousand:

```
balance = 12345.6789;
rounder = 3;
balance = MathLib.round(balance, rounder);
// The value of balance is now 12000.0000
```

In the next example, a rounder value of -2 is used to round balance to two decimal places:

```
balance = 12345.6789;
rounder = -2;
balance = mathLib.round(balance, rounder);
// The value of balance is now 12345.6800
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library MathLib” on page 966

## sin()

The system function **MathLib.sin** that returns the sine of a number. The result is in the range of -1 to 1.

```
MathLib.sin(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

### *result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.sin** function is converted to the format of *result* and returned in *result*.

### *numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

### Example:

```
result = MathLib.sin(myItem);
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library MathLib” on page 966

## sinh()

The system function **MathLib.sinh** returns the hyperbolic sine of a number.

```
MathLib.sinh(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

### *result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.sinh** function is converted to the format of *result* and returned in *result*.



*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

**Example:**

```
result = MathLib.sinh(myItem);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**sqrt()**

The math function **MathLib.sqrt** returns the square root of a number. The function operates on any number that is greater than or equal to zero.

**MathLib.sqrt**(*numericField* **mathLibNumber** *in*)  
returns (*result* **mathLibTypeDependentResult**)

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.sqrt** function is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

**Example:**

```
result = MathLib.sqrt(myItem);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library MathLib” on page 966

**stringAsDecimal()**

The system function **MathLib.stringAsDecimal** accepts a character value (like “98.6”) and returns the equivalent value of type DECIMAL. If the input is non-numeric, the function returns a NULL.

**MathLib.stringAsDecimal**(*numberAsText* **STRING** *in*)  
returns (*result* **DECIMAL** nullable)

*result*

A field that can accept a value of type DECIMAL. The receiving field must be nullable and can have any decimal position and any length.

EGL allows as many as 32 digits on either side of the decimal point. The decimal point (if any) is specific to the Java locale.

For details on the implications of assigning numeric values to fields of different types, see *Assignments*.

*numberAsText*

A character field or literal string, which can include an initial sign character.

**Example:**

```
myField = "-5.243";

// result = -5.243
result = MathLib.stringAsDecimal(myField);
```

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"Assignments" on page 456

"EGL library MathLib" on page 966

### stringAsFloat()

The system function **MathLib.stringAsFloat** accepts a character value (like "98.6") and returns the equivalent value of type FLOAT. If the input is nonnumeric, the function returns a NULL.

**MathLib.stringAsFloat**(*numberAsText* **STRING** in)  
returns (*result* **FLOAT** nullable)

#### *result*

A field that can accept a value of type FLOAT. The receiving field must be nullable and can have any decimal position and any length. The decimal point (if any) is specific to the Java locale.

For details on the implications of assigning numeric values to fields of different types, see *Assignments*.

#### *numberAsText*

A character field or literal string, which can include an initial sign character.

#### Example:

```
myField = "-5.243";

// result = -5.243
result = MathLib.stringAsFloat(myField);
```

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"Assignments" on page 456

"EGL library MathLib" on page 966

### stringAsInt()

The system function **MathLib.stringAsInt** accepts a character value (like "98") and returns the equivalent value of type BIGINT. If the input is nonnumeric, the function returns a NULL.

**MathLib.stringAsInt**(*numberAsText* **STRING** in)  
returns (*result* **BIGINT** nullable)

#### *result*

A field that can accept a value of type BIGINT. The receiving field must be nullable.

For details on the implications of assigning numeric values to fields of different types, see *Assignments*.

#### *numberAsText*

A character field or literal string, which can include an initial sign character.

**Example:**

```
myField = "-5";

// result = -5
result = MathLib.stringAsInt(myField);
```

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"EGL library MathLib" on page 966

**tan()**

The system function **MathLib.tan** returns the tangent of a number.

```
MathLib.tan(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.tan** function is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

**Example:**

```
result = MathLib.tan(myItem);
```

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"EGL library MathLib" on page 966

**tanh()**

The system function **MathLib.tanh** returns the hyperbolic tangent of a number. The result is in the range of -1 to 1.

```
MathLib.tanh(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

*result*

Any numeric or HEX item, as described in *Mathematical (system words)*. The value returned by the **MathLib.tanh** function is converted to the format of *result* and returned in *result*.

*numericField*

Any numeric or HEX item, as described in *Mathematical (system words)*. The item is converted to double-precision floating-point before *result* is calculated.

**Example:**

```
result = MathLib.tanh(myItem);
```

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"EGL library MathLib" on page 966

## recordName.resourceAssociation

When your program does an I/O operation against a record, the I/O is done on the physical file whose name is in the record-specific variable *recordName.resourceAssociation*, if that variable supports the particular type of file. The variable is initialized in accordance with the resourceAssociation part used at generation time; for details, see *Resource associations and file types*. You can change the system resource name at run time by placing a different value in **resourceAssociation**.

In most cases, you must use the syntax *recordName.resourceAssociation*. You do not need to specify a record name, however, if EGL can determine the record that you intended, as is true in each of these cases:

- I/O is only performed against one record in the program
- **resourceAssociation** is used in a function that performs I/O against only one record
- I/O is performed against multiple records in the program, but all records have the same file name; in this case, the first record that appears as an I/O object is used as the implicit qualifier.

You can use **resourceAssociation** as any of the following:

- The source or target operand of an assignment statement
- An item in a logical expression in a **case**, **if**, or **while** statement
- The argument in a **return** or **exit** statement

The characteristics of **resourceAssociation** are as follows:

### Primitive type

CHAR

### Data length

Varies by file type

### Saved across segment?

Yes

## Definition considerations

The value moved into *recordName.resourceAssociation* must be a valid system resource name for the system and file type that were specified when the program was generated. If more than one record specifies the same file name, modification of **resourceAssociation** for any record with that file name changes the setting of **resourceAssociation** for all records in the program with the same file name.

If a system resource identified in the setting of **resourceAssociation** is open when that record-specific variable is modified, the system resource that *was* in that variable is closed in the following circumstance: an I/O option runs against a record that has the same EGL file name as the record that qualifies **resourceAssociation**.

If two programs are using the same EGL file name, each of the record-specific **resourceAssociation** variables must contain the same value. Otherwise the previously opened system resource is closed when a new one is opened.

A comparison of **resourceAssociation** with another value tests true only if the match is exact. If you initialize **resourceAssociation** with a lowercase value, for example, the lowercase value matches only a lowercase value.

**Files shared across programs:** You can set the system resource name either at generation or at runtime:

**At generation time**

If two programs in the same run unit access the same logical file, you must specify the same system resource name for the logical file at generation to ensure that both programs access the same physical file at run time.

**At run time**

If you use *recordName.resourceAssociation*, each program that accesses the file must set **resourceAssociation** for the file. If two programs in the same run unit access the same logical file, each program must set **resourceAssociation** to the same system resource name to ensure that both programs access the same physical file at run time.

If a system resource is shared by multiple programs, each program that accesses the resource must set **resourceAssociation** to refer to the same resource. Also, if two programs in the same run unit access the same logical file, each program must set **resourceAssociation** to the same system resource name at generation time to ensure that both programs access the same system resource at run time.

**MQ records:** The system resource name for MQ records defines the queue manager name and queue name. Specify the name in the following format:

*queueManagerName:queueName*

*queueManagerName*

Name of the queue manager.

*queueName*

Name of the queue.

As shown, the names are separated with a colon. However, *queueManagerName* and the colon can be omitted. The system resource name is used as the initial value for the record-specific **resourceAssociation** item and identifies the default queue associated with the record. For further details, see *MQSeries support*.

**Example**

```
if (process == 1)
  myrec.resourceAssociation = "myFile.txt";
else
  myrec.resourceAssociation = "myFile02.txt";
end
```

**Related concepts**

“MQSeries support” on page 336

“Resource associations and file types” on page 393

**Related reference**

## EGL library ServiceLib

The next table shows the system functions in the library **ServiceLib**. Each argument called *variable* is a variable based either on a Service part or on an Interface part that provides access to a service.

System function and invocation	Description
<i>result</i> = getTCPIPLocation ( <i>variable</i> )	Returns the host name and port number that provide access to an EGL service by way of TCP/IP

System function and invocation	Description
<i>result</i> = <code>getWebEndPoint</code> ( <i>variable</i> )	Returns the URL that provides access to a Web service
<code>setTCPIPLocation</code> ( <i>variable</i> , <i>string</i> )	Sets the host name and port number that provide access to an EGL service by way of TCP/IP
<code>setWebEndPoint</code> ( <i>variable</i> , <i>string</i> )	Sets the URL that provides access to a Web service

### Related concepts

“EGL interfaces” on page 151

“EGL services and Web services” on page 158

### Related tasks

“Creating an EGL Interface part” on page 150

“Creating an Interface part from a Service part” on page 154

“Creating an EGL Service part” on page 157

## getTCPIPLocation()

The ServiceLib function **getTCPIPLocation** returns the host name and port number that provide access to an EGL service by way of TCP/IP.

**ServiceLib.getTCPIPLocation**(*variable* **ServiceOrInterface** in)  
returns (*string* **String**)

#### *variable*

A variable that your code uses to access the service. The variable may be of type Service or Interface. In the latter case, the interface type must be BasicInterface.

#### *string*

The returned string, whose format is as follows:

`host:portNumber`

*host* is the TCP/IP host name that refers to the machine where the service runs.  
*portNumber* is the number of the TCP/IP port that provides access to the service.

The format (including the intervening colon) is equivalent to that of field **tcpipLocation** in the complex property **@EGLBinding**. You can specify that property when defining a Service part.

An exception of type `ServiceBindingException` occurs if the service has a different kind of binding than is required by the function.

### Related concepts

“EGL interfaces” on page 151

“EGL services and Web services” on page 158

### Related tasks

“Creating an EGL Interface part” on page 150

“Creating an EGL Service part” on page 157

“Creating an Interface part from a Service part” on page 154

### Related reference

"EGL library ServiceLib" on page 986

"EGL system exceptions" on page 587

"setTCPIPLocation()"

### getWebEndPoint()

The ServiceLib function **getWebEndPoint** returns the URL that provides access to a Web service.

**ServiceLib.getWebEndPoint**(*variable* **ServiceOrInterface** in)  
returns (**URL String**)

#### *variable*

A variable that your code uses to access the service. The variable may be of type Service or Interface. In the latter case, the interface type must be BasicInterface.

#### *URL*

The returned URL. The format is equivalent to that of the field **endpoint** in the complex property **@WebBinding**, as used for Service parts. Here is an example:

*http://www.ibm.com/myService*

An exception of type ServiceBindingException occurs if the service has a different kind of binding than is required by the function.

### Related concepts

"EGL interfaces" on page 151

"EGL services and Web services" on page 158

### Related tasks

"Creating an EGL Interface part" on page 150

"Creating an EGL Service part" on page 157

"Creating an Interface part from a Service part" on page 154

### Related reference

"EGL library ServiceLib" on page 986

"EGL system exceptions" on page 587

"setWebEndPoint()" on page 989

### setTCPIPLocation()

The ServiceLib function **setTCPIPLocation** sets the host name and port number that provide access to an EGL service by way of TCP/IP.

**ServiceLib.setTCPIPLocation**(*variable* **ServiceOrInterface** in,  
*string* **String** in)

#### *variable*

A variable that your code uses to access the service. The variable may be of type Service or Interface. In the latter case, the interface type must be BasicInterface.

#### *string*

The format of the string that sets the value is as follows:

*host:portNumber*

*host* is the TCP/IP host name that refers to the machine where the service runs.  
*portNumber* is the number of the TCP/IP port that provides access to the service.

The format (including the intervening colon) is equivalent to that of field **tcpipLocation** in the complex property **@EGLBinding**. You can specify that property when defining a Service part.

An exception of type `ServiceBindingException` occurs if the service has a different kind of binding than is required by the function.

#### Related concepts

"EGL interfaces" on page 151

"EGL services and Web services" on page 158

#### Related tasks

"Creating an EGL Interface part" on page 150

"Creating an EGL Service part" on page 157

"Creating an Interface part from a Service part" on page 154

#### Related reference

"EGL library `ServiceLib`" on page 986

"EGL system exceptions" on page 587

"`getTCPIPLocation()`" on page 987

### setWebEndPoint()

The `ServiceLib` function **setWebEndPoint** sets the URL that provides access to a Web service.

```
ServiceLib.setWebEndPoint(variable ServiceOrInterface in,  
string, STRING in)
```

#### *variable*

A variable that your code uses to access the service. The variable may be of type `Service` or `Interface`. In the latter case, the interface type must be `BasicInterface`.

#### *string*

The URL. The format is equivalent to that of the field **endpoint** in the complex property **@WebBinding**, as used for Service parts. An example is here:

```
http://www.ibm.com/myService
```

An exception of type `ServiceBindingException` occurs if the service has a different kind of binding than is required by the function.

#### Related concepts

"EGL interfaces" on page 151

"EGL services and Web services" on page 158

#### Related tasks

"Creating an EGL Interface part" on page 150

"Creating an EGL Service part" on page 157

"Creating an Interface part from a Service part" on page 154

#### Related reference

"EGL library `ServiceLib`" on page 986

"EGL system exceptions" on page 587

"`getWebEndPoint()`" on page 988



## EGL library ReportLib

*ReportLib*, the EGL report library, is a system library that establishes a framework containing all of the components that are needed to interact with the JasperReports library. The EGL report library includes the following components:

- Functions, variables, and constants that are used for these purposes:
  - To interact with JasperReports library functions
  - To define, set, and retrieve the data source for a report
  - To export a filled report to different file formats
  - To manipulate the contents of the report and process report data
- Records containing names of files that store the report design, filled report, and exported report.
- The report

The report library includes the following functions:

System function/Invocation	Description
<code>addReportParameter(report, parameterString, parameterValue)</code>	Adds a value to the parameter list of the report
<code>fillReport(report, source)</code>	Fills the report using the specified data source
<code>exportReport(report, format)</code>	Exports the filled report in the specified format
<code>resetReportParameters(report)</code>	Removes all of the parameters used for a particular report

The following functions are invoked only within report handlers:

System function/Invocation	Description
<code>addReportData(rd, dataSetName)</code>	Adds the report data object with the specified name to the current Report Handler.
<code>result = getReportData(dataSetName)</code>	Retrieves the report data record with the specified name. The returned value is of type <code>ReportData</code> .
<code>result = getReportParameter(parameter)</code>	Returns the value of the specified parameter from the report that is being filled.
<code>result = getFieldValue(fieldName)</code>	Returns the value of the specified field value for the row currently being processed. The returned value is of type <code>ANY</code> .
<code>result = getReportVariableValue(variable)</code>	Returns the value of the specified variable from the report that is being filled. The returned value is of type <code>ANY</code> .
<code>setReportVariableValue(variable, value)</code>	Sets the value of the specified variable to the provided value.

The report library also includes the following Java method that you may invoke only from within a JasperReports XML source file:

System function/Invocation	Description
"getDataSource()" on page 994 ( <i>datasource</i> );	Within a report design file, retrieves a previously stored ReportData data record, in the form of a JRDataSource object.

**Note:** If you delete an EGL report, you must remove all references to the report.

#### Related concepts

"EGL report creation process overview" on page 252

"EGL reports overview" on page 251

### addReportData()

**addReportData()** associates data (as stored in a variable of type ReportData) with an ID. You may retrieve the data in two ways:

- By invoking **ReportLib.getReportData()**
- By invoking the report handler method **getDataSource()** in the design file

```
ReportLib.addReportData(
    rd ReportData in,
    dataID STRING in)
```

*rd* A variable of type reportData

*dataID*

An arbitrary name that you can use to access the data in the variable

#### Related concepts

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

"Syntax diagram for EGL functions" on page 884

#### Related reference

"addReportParameter()"

"EGL library ReportLib" on page 990

"exportReport()" on page 992

"fillReport()" on page 993

### addReportParameter()

The syntax diagram for the **ReportLib.addReportParameter** function is as follows:

```
ReportLib.addReportParameter(
    report Report in,
    parameterString STRING in,
    parameterValue any in)
```

*report*

The name of the report

*parameterString*

The name of the parameter

*parameterValue*

The value of the parameter

Before filling a report, EGL can pass a set of parameters that either establish values to be used in the report or override parameters specified in the XML report design. The **ReportLib.addReportParameter** function adds the value of the specified parameter to the parameter list of the report.

**Note:** See JasperReports documentation for information on JasperReports parameters and data types.

#### Related concepts

“Syntax diagram for EGL functions” on page 884  
EGL report overview  
EGL report creation process overview

#### Related reference

EGL report library  
ReportLib.fillReport function  
ReportLib.exportReport function  
ReportLib.resetReportParameters function

### exportReport()

The system function **ReportLib.exportReport** exports the filled report in the format you specify.

The following diagram illustrates the syntax of that function:

```
ReportLib.exportReport(  
  report Report in,  
  format ExportFormat in)
```

*report*

The report being exported.

*format*

The format and file extension of the exported report.

The values are of the enumeration **ExportFormat**:

**csv**

The output show one value separated from the next with a comma; **csv** stands for comma-separated values.

**html**

The output is in HTML format.

**pdf**

The output is in Adobe Acrobat PDF format.

**text**

The output is in ASCII text format.

#### Related concepts

“EGL reports overview” on page 251  
“EGL report creation process overview” on page 252  
“Enumerations in EGL” on page 578  
“Syntax diagram for EGL functions” on page 884

#### Related tasks

“Exporting reports” on page 274

#### Related reference

“addReportParameter()” on page 991  
“EGL library ReportLib” on page 990  
“fillReport()” on page 993  
“resetReportParameters()” on page 996

## fillReport()

The syntax diagram for the **ReportLib.fillReport** function is as follows:

```
ReportLib.fillReport(  
    report Report in,  
    source DataSource in)
```

*report*

The report to be filled with data.

*source*

The source of the data that is used to fill the report.

Consider this example, which shows how a variable of type **reportData** is associated with the report:

```
eglReport    Report;  
eglReportData ReportData;  
eglReport.reportData = eglReportData;
```

*source* indicates which field to use in the variable of type **ReportData**. Each value of *source* is not a field name, but a value in the enumeration **DataSource**:

### databaseConnection

Use the variable that is referenced in the **connectionName** field of the **reportData** variable, as in this example:

```
eglReportData.connectionName = "mycon";
```

In this case, the SQL statement that accesses data is in the report design file, which is created outside of EGL.

### reportData

Use the variable that is referenced in the **data** field of the **reportData** variable, as in this example:

```
// an array of records, with data  
myRecords customerRecord[];  
  
eglReportData.data = myRecords;
```

### sqlStatement

Use the SQL statement identified in the **sqlStatement** field of the **reportData** variable, as in this example:

```
mySQLString = "Select * From MyTable";  
eglReportData.sqlStatement = mySQLString;
```

Following is an example invocation:

```
ReportLib.fillReport (eglReport, DataSource.sqlStatement);
```

### Related concepts

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

"Enumerations in EGL" on page 578

"Syntax diagram for EGL functions" on page 884

### Related reference

"EGL library ReportLib" on page 990

"addReportParameter()" on page 991

"exportReport()" on page 992

"resetReportParameters()" on page 996

## getDataSource()

getDataSource(), unlike other functions in ReportLib, is a Java method rather than an EGL function. You may call it only from within a JasperReports design file. getDataSource() retrieves previously stored ReportData in form of a JRDataSource object. You may then pass this data source to a subreport.

**ReportLib.getDataSource**(*dataID* **STRING** *in*)  
returns (*data\_source* **JRDataSource**)

*dataID*

A name assigned to a ReportData record during an invocation of **ReportLib.addReportData()**

*data\_source*

A reference to an object of type JRDataSource

### Related concepts

"Syntax diagram for EGL functions" on page 884

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

### Related reference

"addReportData()" on page 991

"addReportParameter()" on page 991

"EGL library ReportLib" on page 990

"exportReport()" on page 992

"fillReport()" on page 993

## getFieldValue()

The **ReportLib.getFieldValue** function returns the value of the specified field for the row currently being processed.

**ReportLib.getFieldValue**(*fieldName* **STRING** *in*)  
returns (*result* **ANY**)

*result*

The value of the specified field

*fieldName*

The name of the specified field

### Related concepts

"Syntax diagram for EGL functions" on page 884

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

### Related reference

"addReportParameter()" on page 991

"fillReport()" on page 993

"EGL library ReportLib" on page 990

"exportReport()" on page 992

## getReportData()

The system function **ReportLib.getReportData** retrieves the report data by using a name specified in **ReportLib.addReportData**.

**ReportLib.getReportData**(*dataID* **STRING** *in*)  
returns (*result* **ReportData**)

*result*

The value of type ReportData

*dataID*

A name assigned to the variable during an invocation of  
**ReportLib.addReportData**

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

#### **Related reference**

"addReportParameter()" on page 991

"EGL library ReportLib" on page 990

"exportReport()" on page 992

"fillReport()" on page 993

### **getReportParameter()**

The **ReportLib.getReportParameter** function returns the value of the specified parameter from the report that is being filled.

**ReportLib.getReportParameter**(*parameter* **STRING** in)  
returns (*result* **ANY**)

*result*

The value of the parameter

*parameter*

The name of the parameter

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

#### **Related reference**

"EGL library ReportLib" on page 990

"addReportParameter()" on page 991

"fillReport()" on page 993

"exportReport()" on page 992

### **getReportVariableValue()**

The system function **ReportLib.getReportVariableValue** returns the value of the specified variable from the report that is being filled.

**ReportLib.getReportVariableValue**(*variable* **STRING** in)  
returns (*result* **ANY**)

*result*

The returned value

*variable*

The variable of interest

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"EGL reports overview" on page 251

"EGL report creation process overview" on page 252

#### Related reference

“addReportParameter()” on page 991  
“EGL library ReportLib” on page 990  
“exportReport()” on page 992  
“fillReport()” on page 993

### resetReportParameters()

The syntax diagram for the **ReportLib.resetReportParameters** function is as follows:

```
ReportLib.resetReportParameters(report Report in)
```

*report*

The name of the report that contains the parameters you want to remove.

The **ReportLib.resetReportParameters** function removes all of the EGL parameters used for a particular report.

#### Related concepts

“EGL reports overview” on page 251  
“EGL report creation process overview” on page 252  
“Syntax diagram for EGL functions” on page 884

#### Related reference

“addReportParameter()” on page 991  
“EGL library ReportLib” on page 990  
“exportReport()” on page 992  
“fillReport()” on page 993

### setReportVariableValue()

The **ReportLib.setReportVariableValue** function sets the value of the specified variable to a value provided to the function.

```
ReportLib.setReportVariableValue(  
  variable STRING in,  
  value Any in)
```

*variable*

The variable to set

*value*

The value to assign

#### Related concepts

“EGL report creation process overview” on page 252  
“EGL reports overview” on page 251  
“Syntax diagram for EGL functions” on page 884

#### Related reference

“addReportParameter()” on page 991  
“EGL library ReportLib” on page 990  
“fillReport()” on page 993  
“exportReport()” on page 992

## EGL library StrLib

The next table shows the system functions in the library **StrLib** and is followed by tables that show the variables and constants in that library.

System function and invocation	Description
<i>result</i> = <code>characterAsInt (text )</code>	Converts a character string into an integer string corresponding to the first character in the character expression.
<i>result</i> = <code>clip (text )</code>	Deletes trailing blank spaces and nulls from the end of returned character strings and can be used to test for NULL.
<i>result</i> = <code>compareStr (target, targetSubstringIndex, targetSubstringLength, source, sourceSubstringIndex, sourceSubstringLength)</code>	Compares two substrings in accordance with their ASCII or EBCDIC order at run time and returns a value (-1, 0, or 1) to indicate which is greater.
<i>result</i> = <code>concatenate (target , source)</code>	Concatenates <i>target</i> and <i>source</i> ; places the new string in <i>target</i> ; and returns an integer that indicates whether <i>target</i> was long enough to contain the new string
<i>result</i> = <code>concatenateWithSeparator (target, source, separator)</code>	Concatenates <i>target</i> and <i>source</i> , inserting <i>separator</i> between them; places the new string in <i>target</i> ; and returns an integer that indicates whether <i>target</i> was long enough to contain the new string
<code>copyStr (target, targetSubstringIndex, targetSubstringLength, source, sourceSubstringIndex, sourceSubstringLength)</code>	Copies one substring to another
<i>result</i> = <code>findStr (source, sourceSubstringIndex, sourceSubstringLength, searchString)</code>	Searches for the first occurrence of a substring within a string
<i>result</i> = <code>formatDate (dateValue [, dateFormat])</code>	Formats a date value and returns a value of type STRING. The default format is the format specified in the current locale.
<i>result</i> = <code>formatNumber (numericExpression, numericFormat)</code>	Returns a number as a formatted string.
<i>result</i> = <code>formatTime (timeValue [, timeFormat])</code>	Formats a parameter into a time value and returns a value of type STRING. The default format is the format specified in the current locale.
<i>result</i> = <code>formatTimeStamp (timestampValue [, timestampFormat])</code>	Formats a parameter into a timestamp value and returns a value of type STRING. The DB2 format is the default format.
<i>result</i> = <code>getNextToken (target, source, sourceSubstringIndex, sourceStringLength, characterDelimiter)</code>	Searches a string for the next token and copies the token to <i>target</i>
<i>result</i> = <code>integerAsChar (integer)</code>	Converts an integer string into a character string.
<i>result</i> = <code>lowerCase (text)</code>	Converts all uppercase values in a character string to lowercase values. Numeric and existing lowercase values are not affected.
<code>setBlankTerminator (target)</code>	Replaces a null terminator and any subsequent characters in a string with spaces, so that a string value returned from a C or C++ program can operate correctly in an EGL-generated program
<code>setNullTerminator (target)</code>	Changes all trailing spaces in a string to nulls



System function and invocation	Description
<code>setSubStr (target, targetSubstringIndex, targetSubstringLength, source)</code>	Replaces each character in a substring with a specified character
<code>result =spaces (characterCount)</code>	Returns a string of a specified length.
<code>result = strLen (source)</code>	Returns the number of bytes in an item, excluding any trailing spaces or nulls
<code>result = textLen (source)</code>	Returns the number of bytes in a text expression, excluding any trailing spaces or nulls
<code>result = upperCase (characterItem)</code>	Converts all lowercase values in a character string to uppercase values. Numeric and existing uppercase values are not affected.

The next table shows the system variables in the library **StrLib**.

System variable	Description
<code>defaultDateFormat</code>	Specifies the value of <b>defaultDateFormat</b> , which is one of several masks that can be used to create the string returned by the function <b>StrLib.formatDate</b> .
<code>defaultMoneyFormat</code>	Specifies the value of <b>defaultMoneyFormat</b> , which is one of several masks that can be used to create the string returned by the function <b>StrLib.formatNumber</b> .
<code>defaultNumericFormat</code>	Specifies the value of <b>defaultNumericFormat</b> , which is one of several masks that can be used to create the string returned by the function <b>StrLib.formatNumber</b> .
<code>defaultTimeFormat</code>	Specifies the value of <b>defaultTimeFormat</b> , which is one of several masks that can be used to create the string returned by the function <b>StrLib.formatTime</b> .
<code>defaultTimestampFormat</code>	Specifies the value of <b>defaultTimestampFormat</b> , which is one of several masks that can be used to create the string returned by the function <b>StrLib.formatTimestamp</b> .

The next table shows the system constants in the library **StrLib**. All are of type **STRING**.

System constant	Description
<code>db2TimestampFormat</code>	The pattern <i>yyyy-MM-dd-HH.mm.ss.ffffff</i> , which is the IBM DB2 default timestamp format.
<code>eurDateFormat</code>	The pattern <i>dd.MM.yyyy</i> , which is the IBM European standard date format.
<code>eurTimeFormat</code>	The pattern <i>HH.mm.ss</i> , which is the IBM European standard time format.

System constant	Description
isoDateFormat	The pattern <i>yyyy-MM-dd</i> , which is the date format specified by the International Standards Organization (ISO).
isoTimeFormat	The pattern <i>HH:mm:ss</i> , which is the time format specified by the International Standards Organization (ISO).
jisDateFormat	The pattern <i>yyyy-MM-dd</i> , which is the Japanese Industrial Standard date format.
jisTimeFormat	The pattern <i>HH:mm:ss</i> , which is the Japanese Industrial Standard time format.
odbcTimestampFormat	The pattern <i>yyyy-MM-dd HH:mm:ss.ffffff</i> , which is the ODBC timestamp format.
usaDateFormat	The pattern <i>MM/dd/yyyy</i> , which is the IBM USA standard date format.
usaTimeFormat	The pattern <i>hh:mm AM</i> , which is the IBM USA standard time format.

### Related reference

[“formatDate\(\)” on page 1007](#)  
[“formatNumber\(\)” on page 1007](#)  
[“formatTime\(\)” on page 1008](#)  
[“formatTimeStamp\(\)” on page 1009](#)

## characterAsInt()

The string-formatting function **StrLib.characterAsInt** converts a character string into an integer string corresponding to the first character in the character expression.

```
StrLib.characterAsInt(text STRING in)
returns (result INT)
```

*result*

A variable of type INT.

*text*

A literal, variable, or expression that returns a character string of type CHAR.

To convert an integer string into a character string, use the **StrLib.integerAsChar** string-formatting function.

### Related reference

[“EGL library StrLib” on page 996](#)  
[“integerAsChar\(\)” on page 1012](#)

## clip()

The string-formatting function **StrLib.clip** deletes trailing blank spaces and nulls from the end of returned strings and can be used to test for NULL.

```
StrLib.clip(text STRING in)
returns (result STRING)
```

*result*

A character string.

*text*

A literal, variable, or expression that returns a character string of type CHAR.

If the build descriptor option **itemsNullable** is set to YES or if the field-level property **isNullable** is set to YES for an SQL record field, you can set NULL by assigning an empty string to the field; and you can test NULL by testing the result of passing the string to the function StrLib.clip:

```
myString String = "";

// indicates that the variable is NULL
if (StrLib.clip(myString) is NULL)
;
end
```

### Related reference

“EGL library StrLib” on page 996

## compareStr()

The system function **StrLib.compareStr** compares two substrings in accordance with their ASCII or EBCDIC order at run time.

```
StrLib.compareStr(
  target a character type in,
  targetSubstringIndex INT in,
  targetSubstringLength INT in,
  source a character type in,
  sourceSubstringIndex INT in,
  sourceSubstringLength INT in )
returns (result INT)
```

*result*

Numeric field that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

- 1      The substring based on *target* is less than the substring based on *source*
- 0       The substring based on *target* is equal to the substring based on *source*
- 1       The substring based on *target* is greater than the substring based on *source*

*target*

String from which a target substring is derived. Can be a field or a literal.

*targetSubStringIndex*

Identifies the starting byte of the substring in *target*, given that the first byte in *target* has the index value 1. This index can be an integer literal. Alternatively, this index can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*targetSubStringLength*

Identifies the number of bytes in the substring that is derived from *target*. The length can be an integer literal. Alternatively, this index can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*source*

String from which a source substring is derived. Can be a field or a literal.

*sourceSubStringIndex*

Identifies the starting byte of the substring in *source*, given that the first byte in *source* has the index value of 1. This index can be an integer literal.

Alternatively, this index can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*sourceSubStringLength*

Identifies the number of bytes in the substring that is derived from *source*. The length can be an integer literal. Alternatively, this index can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

A byte-to-byte binary comparison of the substring values is performed. If the substrings are not the same length, the shorter substring is padded with spaces before the comparison.

**Definition considerations:** The following values are returned in **sysVar.errorCode**:

- 8 Index less than 1 or greater than string length.
- 12 Length less than 1.
- 20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character
- 24 Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

**Example:**

```
target = "123456";
source = "34";
result =
  StrLib.compareStr(target,3,2,source,1,2);
// result = 0
```

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"compareBytes()" on page 1050

"EGL library StrLib" on page 996

**concatenate()**

The system function **StrLib.concatenate** concatenates two strings.

```
StrLib.concatenate(
  target a character type inOut,
  source a character type in)
returns (result INT)
```

*result*

Numeric field that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

- 1 Concatenated string is too long to fit in the target field and the string was truncated, as described later
- 0 Concatenated string fits in the target field

*target*

Target field

*source*

Source field or literal

When two strings are concatenated, the following occurs:

1. Any trailing spaces or nulls are deleted from the target string.
2. The source string is appended to the string produced by the previous step.
3. If the output produced by the second step is longer than the target string field, the output is truncated. If the output is shorter than the target field, the output is padded with blanks.

**Example:**

```
phrase = "and/ "; // CHAR(7)
or      = "or";
result =
  StrLib.concatenate(phrase,or);
if (result == 0)
  print phrase; // phrase = "and/or "
end
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library StrLib” on page 996

**concatenateWithSeparator()**

The system function **StrLib.concatenateWithSeparator** concatenates two strings, inserting a separator string between them. If the initial length of the target string is zero (not counting trailing blanks and nulls), the separator is omitted and the source string is copied to the target string.

```
StrLib.concatenateWithSeparator(
  target VagText inOut,
  source VagText in,
  separator VagText in)
returns (result INT)
```

*result*

Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function: :

- |    |  |
|----|--|
| 0  | Concatenated string fits in target item.   |
| -1 | Concatenated string is too long to fit in the target item and the string was truncated, as described later |

*target*

Target item.

*source*

Source item or literal.

*separator*

Separator item or literal.

Trailing spaces and nulls are truncated from *target*; then, the *separator* string and *source* are appended to the truncated value. If the concatenation is longer than the target allows, truncation occurs. If the concatenation is shorter than the target allows, the concatenated value is padded with spaces.

**Example:**

```

phrase = "and";    // CHAR(7)
or      = "or";
result =
    StrLib.concatenateWithSeparator(phrase,or,"/");
if (result == 0)
    print phrase; // phrase = "and/or "
end

```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library StrLib” on page 996

## copyStr()

The system function **StrLib.copyStr** copies one substring to another.

```

StrLib.copyStr(
    target a character type inOut,
    targetSubstringIndex INT in,
    targetSubstringLength INT in,
    source a character type in,
    sourceSubstringIndex INT in,
    sourceSubstringLength INT in)

```

#### *target*

String from which a target substring is derived. Can be an item or a literal.

#### *targetSubstringIndex*

Identifies the starting byte in *target*, given that the first byte in *target* has the value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

#### *targetSubstringLength*

Identifies the number of bytes in the substring that is derived from *target*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

#### *source*

String from which a source substring is derived. Can be an item or a literal.

#### *sourceSubstringIndex*

Identifies the starting byte of the substring in *source*, given that the first byte in *source* has the value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

#### *sourceSubstringLength*

Identifies the number of bytes in the substring that is derived from *source*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

If the source is longer than the target, the source is truncated. If the source is shorter than the target, the source value is padded on the right with spaces.

**Definition considerations:** The following values are returned in **sysVar.errorCode**:

8            Index less than 1 or greater than string length.

- 12      Length less than 1.
- 20      Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character.
- 24      Invalid double-byte length. Length in bytes for a DBCS or UNICODE string is odd (double-byte lengths must always be even).

**Example:**

```
target = "120056";
source = "34";
StrLib.copyStr(target,3,2,source,1,2);
// target = "123456"
```

**Related concepts**

"Syntax diagram for EGL functions" on page 884

**Related reference**

"EGL library StrLib" on page 996

### **defaultDateFormat (EGL system variable)**

The system variable **StrLib.defaultDateFormat** specifies one of several masks that can be used to create the string returned by the function **StrLib.formatDate**.

The initial value of **StrLib.defaultDateFormat** is the value of the Java runtime property **vgj.default.dateFormat**. If that property is not set, the initial value of **StrLib.defaultDateFormat** is *MM/dd/yyyy*.

For details on the characteristics of a time mask, see *Date, time, and timestamp specifiers*.

Type: STRING

**Related reference**

"Date, time, and timestamp format specifiers" on page 46

"EGL library StrLib" on page 996

"formatDate()" on page 1007

"Java runtime properties (details)" on page 642

### **defaultMoneyFormat (EGL system variable)**

The system variable **StrLib.defaultMoneyFormat** specifies one of several masks that can be used to create the string returned by the function **StrLib.formatNumber**.

The initial value of **StrLib.defaultMoneyFormat** is the value of the Java runtime property **vgj.default.moneyFormat**. If that property is not set, the initial value of **StrLib.defaultMoneyFormat** is an empty string.

For details on the characteristics of a numeric mask, see *formatNumber()*.

Type: STRING

**Related reference**

"EGL library StrLib" on page 996

"formatNumber()" on page 1007

"Java runtime properties (details)" on page 642

## defaultNumericFormat (EGL system variable)

The system variable **StrLib.defaultNumericFormat** specifies one of several masks that can be used to create the string returned by the function **StrLib.formatNumber**.

The initial value of **StrLib.defaultNumericFormat** is the value of the Java runtime property **vgj.default.numericFormat**. If that property is not set, the initial value of **StrLib.defaultNumericFormat** is an empty string.

For details on the characteristics of a numeric mask, see *formatNumber()*.

Type: STRING

### Related reference

"EGL library StrLib" on page 996

"formatNumber()" on page 1007

"Java runtime properties (details)" on page 642

## defaultTimeFormat (system variable)

The system variable **StrLib.defaultTimeFormat** specifies one of several masks that can be used to create the string returned by the function **StrLib.formatTime**. The variable is not used in any other context.

The initial value of **StrLib.defaultTimeFormat** is the value of the Java runtime property **vgj.default.timeFormat**. If that property is not set, the initial value of **StrLib.defaultTimeFormat** is *HH:mm:ss*.

For details on the characteristics of a time mask, see *Date, time, and timestamp specifiers*.

Type: STRING

### Related reference

"Date, time, and timestamp format specifiers" on page 46

"EGL library StrLib" on page 996

"formatTime()" on page 1008

"Java runtime properties (details)" on page 642

## defaultTimestampFormat (EGL system variable)

The system variable **StrLib.defaultTimestampFormat** specifies one of several masks that can be used to create the string returned by the function **StrLib.formatTimestamp**.

The initial value of **StrLib.defaultTimestampFormat** is the value of the Java runtime property **vgj.default.timestampFormat**. If that property is not set, the initial value of **StrLib.defaultTimestampFormat** is an empty string.

For details on the characteristics of a timestamp mask, see *Date, time, and timestamp specifiers*.

Type: STRING



### Related reference

"Date, time, and timestamp format specifiers" on page 46

"EGL library StrLib" on page 996

"formatTimeStamp()" on page 1009

"Java runtime properties (details)" on page 642

### findStr()

The system function **StrLib.findStr** searches for the first occurrence of a substring in a string.

```
StrLib.findStr(  
    source VagText in,  
    sourceSubstringIndex INT inOut,  
    sourceSubstringLength INT in,  
    searchString VagText in)  
returns (result INT)
```

#### result

Numeric item that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

- 1      Search string was not found
- 0       Search string was found

#### source

String from which a source substring is derived. Can be an item or a literal.

#### sourceSubstringIndex

Identifies the starting byte for the substring in *source*, given that the first byte in *source* has the index value of 1. This index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

#### sourceStringLength

Identifies the number of bytes in the substring that is derived from *source*. This index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

#### searchString

String item or literal to be searched for in the source substring. Trailing blanks or nulls are truncated from the search string before searching begins.

If *searchString* is found in the source substring, *sourceSubstringIndex* is set to indicate its location (the byte of the source where the matching substring begins). Otherwise, *sourceSubstringIndex* is not changed.

**Definition considerations:** The following values are returned in `sysVar.errorCode`:

- 8        Index less than 1 or greater than string length.
- 12      Length less than 1.
- 20      Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character.
- 24      Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

### Example:

```
source = "123456";  
sourceIndex = 1  
sourceLength = 6
```

```
search = "34";
result =
  StrLib.findStr(source,sourceIndex,sourceLength,"34");
// result = 0, sourceIndex = 3
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library StrLib” on page 996

### formatDate()

The system function **StrLib.formatDate** formats a date value and returns a value of type STRING.

```
StrLib.formatDate(
  dateValue DATE in
  [, dateFormat STRING in])
returns (result STRING)
```

*result*

A variable of type STRING.

*dateValue*

The value to be formatted. Can be any expression that resolves to a date value; for example, the system variable **VGVar.currentGregorianCalendar**.

*dateFormat*

Identifies the date format, as described in *Date, time, and timestamp specifiers*. If you specify no value for *dateFormat*, EGL run time uses the date format in the Java locale.

You can use a string, the system variable **StrLib.defaultDateFormat** (as described in *defaultDateFormat*), or any of these constants:

#### **StrLib.eurDateFormat**

The pattern *dd.MM.yyyy*, which is the IBM European standard date format

#### **StrLib.isoDateFormat**

The pattern *yyyy-MM-dd*, which is the date format specified by the International Standards Organization (ISO)

#### **StrLib.jisDateFormat**

The pattern *yyyy-MM-dd*, which is the Japanese Industrial Standard date format

#### **StrLib.usaDateFormat**

The pattern *MM/dd/yyyy*, which is the IBM USA standard date format

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“DATE” on page 41

“Date, time, and timestamp format specifiers” on page 46

“defaultDateFormat (EGL system variable)” on page 1004

“EGL library StrLib” on page 996

### formatNumber()

The string function **StrLib.formatNumber** returns a number as a formatted string.

```

StrLib.formatNumber(
    numericExpression anyNumericExpression in
    [ , numericFormat STRING in] )
returns (result STRING)

```

*result*

A variable of type **STRING**.

*numericExpression*

The numeric value to be formatted. Can be any expression that resolves to a number.

*numericFormat*

A string that defines how the number is to be formatted. The string is optional.

You can use the system variable **StrLib.defaultMoneyFormat** or **StrLib.defaultNumericFormat**. For details on those variables, see *defaultMoneyFormat* and *defaultNumericFormat*..

Valid characters are as follows:

- # A placeholder for a digit.
- \* Use an asterisk (\*) as the fill character for a leading zero.
- & Use a zero as the fill character for a leading zero.
- # Use a space as the fill character for a leading zero.
- < Left justify the number.
- , Use a locale-dependent numeric separator unless the position contains a leading zero.
- . Use a locale-dependent decimal point.
- Use a minus sign (-) for values less than 0; use a space for values greater than or equal to 0.
- + Use a minus sign for values less than 0; use a plus sign (+) for values greater than or equal to 0.
- ( Precede negative values with a left parenthesis, as appropriate in accounting.
- ) Place a right parenthesis after a negative value, as appropriate in accounting.
- \$ Precede the value with the locale-dependent currency symbol.
- @ Place the locale-dependent currency symbol after the value.

#### Related reference

“defaultMoneyFormat (EGL system variable)” on page 1004

“defaultNumericFormat (EGL system variable)” on page 1005

“EGL library StrLib” on page 996

### formatTime()

The datetime function **StrLib.formatTime** formats a time value and returns a value of type **STRING**.

```

StrLib.formatTime(
    aTime Time in
    [ , timeFormat STRING in
    ])
returns (result STRING)

```

*result*

A variable of type **STRING**.

*aTime*

The value to be formatted. Can be any expression that resolves to a time value; for example, the system variable **DateTimeLib.currentTime**.

*timeFormat*

Identifies the time format, as described in *Date, time, and timestamp specifiers*. If you specify no value for *timeFormat*, EGL run time uses the time format in the Java locale.

You can use a string, the system variable **StrLib.defaultTimeFormat** (as described in *defaultTimeFormat*), or any of these constants:

**eurTimeFormat**

The pattern *HH.mm.ss*, which is the IBM European standard time format.

**isoTimeFormat**

The pattern *HH.mm.ss*, which is the time format specified by the International Standards Organization (ISO).

**jisTimeFormat**

The pattern *HH:mm:ss*, which is the Japanese Industrial Standard time format.

**usaTimeFormat**

The pattern *hh:mm AM*, which is the IBM USA standard time format.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"Date, time, and timestamp format specifiers" on page 46

"defaultTimeFormat (system variable)" on page 1005

"EGL library StrLib" on page 996

"TIME" on page 44

### formatTimeStamp()

The datetime formatting function **StrLib.formatTimeStamp** formats a parameter into a timestamp value and returns a value of type STRING.

```
StrLib.formatTimeStamp(  
    aTimeStamp TimeStamp in  
    [, timestampFormat STRING in  
    ]  
)  
returns (result STRING)
```

*result*

A variable of type STRING.

*aTimeStamp*

The **TIMESTAMP** value to be formatted. Can be any expression that resolves to a **TIMESTAMP** value; for example, the system variable **DateTimeLib.currentTimeStamp**.

*timestampFormat*

Identifies the date format, as described in *Date, time, and timestamp specifiers*.

You can use a string, the system variable **StrLib.defaultTimestampFormat** (as described in *defaultTimestampFormat*), or one of these constants:

**db2TimeStampFormat**

The pattern *yyyy-MM-dd-HH.mm.ss.ffffff*, which is the IBM DB2 default timestamp format.

### **odbcTimeStampFormat**

The pattern *yyyy-MM-dd HH:mm:ss.ffffff*, which is the ODBC timestamp format.

### **Related concepts**

"Syntax diagram for EGL functions" on page 884

### **Related reference**

"currentTimeStamp()" on page 921

"Date, time, and timestamp format specifiers" on page 46

"defaultTimestampFormat (EGL system variable)" on page 1005

"EGL library StrLib" on page 996

"TIMESTAMP" on page 44

### **getNextToken()**

The system function **StrLib.getNextToken** searches a substring for a token and copies that token to a target item.

Tokens are strings separated by delimiter characters. For example, if the characters space (" ") and comma (",") are defined as delimiters, the string "CALL PROGRAM ARG1,ARG2,ARG3" can be broken down into the five tokens "CALL", "PROGRAM", "ARG1", "ARG2", and "ARG3".

```
StrLib.getNextToken(  
    target VagText inOut,  
    source VagText in,  
    sourceSubstringIndex INT inOut,  
    sourceSubstringLength INT inOut,  
    characterDelimiter VagText in)  
returns (result INT)
```

#### *result*

An item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. The value is one of these:

- +n**      Number of characters in the token. The token is copied from the substring under review to the target item.
- 0**        No token was in the substring under review.
- 1**      The token was truncated when copied to the target item.

#### *target*

Target item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE.

#### *source*

Source item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE. May be a literal of any of those types other than UNICODE.

#### *sourceSubstringIndex*

Identifies the starting byte at which to begin searching for a delimiter, given that the first byte in *source* has the value 1. *sourceSubstringIndex* can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. If a token is found, the value in *sourceSubstringIndex* is changed to the index of the first character that follows the token.

#### *sourceSubstringLength*

Indicates the number of bytes in the substring under review.

*sourceSubstringLength* can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places. If a token is found, the value in *sourceSubstringLength* is changed to the number of bytes in the substring that begins after the returned token.

### *characterDelimiter*

One or more delimiter characters, with no characters separating one from the next. May be an item of type CHAR, DBCHAR, HEX, MBCHAR, or UNICODE. May be a literal of any of those types other than UNICODE.

You can invoke a sequence of calls to retrieve each token in a substring without resetting the values for *sourceSubstringIndex* and *sourceSubstringLength*, as shown in a later example.

**Error conditions:** The following values are returned in **SysVar.errorCode**:

- 8        *sourceSubstringIndex* is less than 1 or is greater than number of bytes in the substring under review.
- 12      *sourceSubstringLength* is less than 1.
- 20      The value in *sourceSubstringIndex* for a DBCHAR or UNICODE string refers to the middle of a double-byte character.
- 24      The value in *sourceSubstringLength* for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

### **Example:**

```
Function myFunction()
  myVar myStructurePart;
  myRecord myRecordPart;

  i = 1;
  myVar.mySourceSubstringIndex = 1;
  myVar.mySourceSubstringLength = 29;

  while (myVar.mySourceSubstringLength > 0)
    myVar.myResult = StrLib.getNextToken( myVar.myTarget[i],
      "CALL PROGRAM arg1, arg2, arg3",
      myVar.mySourceSubstringIndex,
      myVar.mySourceSubstringLength, " ," );

    if (myVar.myResult > 0)
      myRecord.outToken = myVar.myTarget[i];
      add myRecord;
      set myRecord empty;
      i = i + 1;
    end
  end
end

Record myStructurePart
  01 myTarget CHAR(80)[5];
  01 mySource CHAR(80);
  01 myResult myBinPart;
  01 mySourceSubstringIndex INT;
  01 mySourceSubstringLength BIN(9,0);
  01 i myBinPart;
end

Record myRecordPart
  serialRecord:
    fileName="Output"
  end
  01 outToken CHAR(80);
end
```

### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library StrLib” on page 996

### integerAsChar()

The string-formatting function **StrLib.integerAsChar** converts an integer string into a character string.

```
StrLib.integerAsChar(integer INT in)  
returns (result STRING)
```

*result*

A variable of type STRING.

*integer*

A literal, variable or expression that returns an integer of type BIGINT, INT or SMALLINT.

To convert a character string into an integer string, use the **StrLib.characterAsInt** string-formatting function.

#### Related reference

“EGL library StrLib” on page 996

“AIBTDLI()” on page 930

### lowerCase()

The string-formatting function **StrLib.lowerCase** converts all uppercase values in a character string to lowercase values. Numeric and existing lowercase values are not affected.

```
StrLib.lowerCase(text STRING in)  
returns (result STRING)
```

*result*

A variable of type STRING.

*text*

A literal, variable, or expression that returns a character string of type CHAR.

The **StrLib.lowerCase** function has no effect on double-byte characters in items of type DBCHAR or MBCHAR.

To convert lowercase values to uppercase values, use the **StrLib.upperCase** string-formatting function.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library StrLib” on page 996

“upperCase()” on page 1015

### setBlankTerminator()

The system function **StrLib.setBlankTerminator** changes a null terminator and any subsequent characters to spaces. **StrLib.setBlankTerminator** changes a string value returned from a C or C++ program to a character value that can operate correctly in an EGL program.

```
StrLib.setBlankTerminator(target VagText inOut)
```

*target*

The target string item. If no null is found in *targetString*, the function has no effect.

**Example:**

```
StrLib.setBlankTerminator(target);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library StrLib” on page 996

**setNullTerminator()**

The system function **StrLib.setNullTerminator** changes all trailing spaces in a string to nulls. You can use **StrLib.setNullTerminator** to convert an item before passing it to a C or C++ program that expects a null-terminated string as an argument.

```
StrLib.setNullTerminator(target VagText inOut)
```

*target*

String to be converted

The target string is searched for trailing spaces and nulls. Any spaces found are changed to nulls.

**Definition considerations:** The following value can be returned in **sysVar.errorCode**:

**16** Last byte of string is not a space or null

**Example:**

```
StrLib.setNullTerminator(myItem01);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library StrLib” on page 996

**setSubStr()**

The system function **StrLib.setSubStr** replaces each character in a substring with a specified character.

```
StrLib.setSubStr(  
    target VagText inOut,  
    targetSubstringIndex INT in,  
    targetSubstringLength INT in,  
    source)
```

*target*

Item that is changed.

*targetSubstringIndex*

Identifies the starting byte of the substring in *target*, given that the first byte in *target* has the index value of 1. This index can be an integer literal.

Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.



*targetSubStringLength*

Identifies the number of bytes in the substring that is derived from *target*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*source*

If the target item is CHAR, MBCHAR, or HEX, the source item must be a one-byte CHAR, MBCHAR, or HEX item or a CHAR literal. If the target is a DBCHAR or UNICODE item, the source must be a single-character DBCHAR or UNICODE item.

**Definition considerations:** The following values are returned in SysVar.errorCode:

- 8 Index less than 1 or greater than string length
- 12 Length less than 1
- 20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character
- 24 Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even)

**Example:**

```
StrLib.setSubStr(target,12,5," ");
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library StrLib” on page 996

## spaces()

The system function **StrLib.spaces** returns a string composed of a specified number of spaces.

```
StrLib.spaces(characterCount INT in)  
returns (result STRING)
```

*result*

A variable of type STRING.

*characterCount*

The length of the string of spaces to be returned.

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library StrLib” on page 996

## strLen()

The system function **StrLib.strLen** returns the number of bytes in an item, excluding any trailing spaces and nulls.

```
StrLib.strLen(source VagText in)  
returns (result INT)
```

*result*

An item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*source*

String item or literal to be measured.

**Example:**

```
length = StrLib.strLen(source);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library StrLib” on page 996

**textLen()**

The system function **StrLib.textLen** returns the number of characters in a text expression, excluding any trailing spaces and nulls.

```
StrLib.textLen(source STRING in)  
returns (result INT)
```

*result*

An item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*source*

The text expression of interest.

**Example:**

```
length = StrLib.textLen(source);
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“EGL library StrLib” on page 996

“Text expressions” on page 601

**upperCase()**

The string-formatting function **StrLib.upperCase** converts all lowercase values in a character string to uppercase values. Numeric and existing uppercase values are not affected.

```
StrLib.upperCase(text STRING in)  
returns (result STRING)
```

*result*

A variable of type STRING.

*text*

A literal, variable, or expression that returns a character string of type CHAR.

The **StrLib.upperCase** function has no effect on double-byte characters in items of type DBCHAR or MBCHAR.

To convert a character string to lowercase, use the **StrLib.lowerCase** string-formatting function.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library StrLib” on page 996

“lowerCase()” on page 1012

## EGL library SysLib

The EGL library SysLib contains a number of functions and a single variable.

Function	Description
<code>beginDatabaseTransaction([database])</code>	Begins a relational-database transaction, but only when the EGL runtime is not committing changes automatically.
<code>result = bytes(field)</code>	Returns the number of bytes in a named area of memory.
<code>calculateChkDigitMod10 (text, checkLength, result)</code>	Places a modulus-10 check digit in a character item that begins with a series of integers.
<code>calculateChkDigitMod11 (text, checkLength, result)</code>	Places a modulus-11 check digit in a character item that begins with a series of integers.
<code>callCmd (commandString[, modeString])</code>	Runs a system command and waits until the command finishes.
<code>commit()</code>	Saves updates that were made to databases, MQSeries message queues, and CICS recoverable files since the last commit. A generated Java program or wrapper also saves the updates done by a remote, CICS-based COBOL program (including updates to CICS recoverable files), but only when the call to the remote COBOL program involves a client-controlled unit of work, as described in <i>luwControl</i> in <i>callLink</i> element.
<code>result = conditionAsInt (booleanExpression)</code>	Accepts a logical expression (like <i>myVar == 6</i> ), returning a 1 if the expression is true, a 0 if the expression is false.
<code>connect (database, userID, password[, commitScope[, disconnectOption[, isolationLevel[, commitControl]]]])</code>	Closes all cursors, releases locks, ends any existing connection, and connects to the database.
<code>convert (target, direction, conversionTable)</code>	Converts data between EBCDIC (host) and ASCII (workstation) formats or performs code-page conversion within a single format.
<code>defineDatabaseAlias (alias, database)</code>	Creates an alias that can be used to establish a new connection to a database to which your code is already connected.
<code>disconnect ([database])</code>	Disconnects from the specified database or (if no database is specified) from the current database.
<code>disconnectAll ()</code>	Disconnects from all the currently connected databases.

Function	Description
<code>errorLog ()</code>	Copies text into the error log that was started by the system function <b>SysLib.startLog</b> .
<code>result = getCmdLineArg (index)</code>	Returns the specified argument from the list of arguments with which the EGL program was involved. The specified argument is returned as a string value.
<code>result = getCmdLineArgCount ()</code>	Returns the number of arguments that were used to start the main EGL program.
<code>result = getMessage (key [, insertArray])</code>	Returns a message from the file that is referenced in the Java runtime property <code>vgj.message.file</code> .
<code>result = getProperty(propertyName)</code>	Retrieves the value of a Java runtime property. If the specified property is not found, the function returns a null string ( <code>""</code> ).
<code>loadTable (filename, insertintoClause[, delimiter])</code>	Loads data from a file into a relational database.
<code>result = maximumSize (arrayName)</code>	Returns the maximum number of rows that can be in a dynamic array of data items or records; specifically, the function returns the value of the array property <b>maxSize</b> .
<code>queryCurrentDatabase (product, release)</code>	Returns the product and release number of the currently connected database.
<code>rollback ()</code>	Reverses updates that were made to databases and MQSeries message queues since the last commit. That reversal occurs in any EGL-generated application.
<code>setCurrentDatabase (database)</code>	Makes the specified database the currently active one.
<code>setError (itemInError, msgKey[, itemInsert])</code> <code>setError (this, msgKey[, itemInsert])</code> <code>setError (msgText)</code>	Associates a message with an item in a PageHandler or UI record or with the PageHandler or UI record as a whole. The message is placed at the location of a JSF message or messages tag in the JSP and is displayed when the related Web page is displayed.
<code>setLocale (languageCode, countryCode[, variant])</code>	Used in PageHandlers and in programs that run in a Web application.
<code>setRemoteUser (userID, passWord)</code>	Sets the userid and password that are used on calls to remote programs from Java programs.
<code>result = size (arrayName)</code>	Returns the number of rows in the specified data table or the number of elements in the specified array. The array may be a structure-item array, a static array of data items or records, or a dynamic array of data items or records.
<code>startCmd (commandString[, modeString])</code>	Runs a system command and does not wait until the command finishes.
<code>startLog (logFile)</code>	Opens an error log. Text is written into that log every time your program invokes <b>SysLib.errorLog</b> .

Function	Description
<code>startTransaction (termID[, prID[, termID]])</code>	Invokes a main program asynchronously, associates that program with a printer or terminal device, and passes a record. If the receiving program is generated by EGL, the record is used to initialize the input record; if the receiver is produced by VisualAge Generator, the record is used to initialize the working storage.
<code>unloadTable (filename, selectStatement[, delimiter])</code>	Unloads data from a relational database into a file.
<code>verifyChkDigitMod10 (input, checkLength, result)</code>	Verifies a modulus-10 check digit in a character item that begins with a series of integers.
<code>verifyChkDigitMod11 (input, checkLength, result)</code>	Verifies a modulus-11 check digit in a character item that begins with a series of integers.
<code>wait (timeInSeconds0</code>	Suspends execution for the specified number of seconds.
<code>"writeStderr()" on page 1046 (textString0</code>	Writes the text string to stderr (Java) or to the COBOL output device.
<code>"writeStdout()" on page 1046 (textString0</code>	Writes the text string to stdout (Java) or to the COBOL output device.

Variable	Description
<code>currentException</code>	A dictionary that identifies the exception that was thrown most recently in the run unit.

## audit()

The system function **SysLib.audit()** writes tracking information to the system log or journal. A program can call this function in the following environments:

- CICS for z/OS (writes to CICS journal)
- IMS BMP (writes to IMS log)
- IMS/VS (writes to IMS log)
- z/OS batch (with limitations described below)

```
SysLib.audit(
    record anyBasicRecord in
    [, jid SMALLINT in
    ])
```

*record*

The name of a record to be written to a journal file.

The first 2 bytes contain the length of the record to be written. The next 2 bytes contain a code that you specify to identify the source of the journal record, and the first byte of that code must be in the range X'A0' to X'FF'.

In addition to containing the record length and record identifier code, the first 28 bytes are reserved for system usage and should not contain user data. Bytes 29 to 32767 are available for audit data.

*jid* An optional parameter that specifies the ID (1-99) of the journal file to

which the service routine writes the record. If `jid` is omitted, the record is written, by default, to the system journal. The parameter is a 2-byte binary number.

**SysVar.errorCode** receives one of the status codes in the next table if you enclose **SysLib.audit** in a **try...end** block or if you are in VisualAge Generator compatibility mode and **VGVar.handleSysLibraryErrors** is set to 1.

Value in SysVar.errorCode	Description
00000000	Successful completion
00000201	Length error
00000202	User source code error
00000204	Journal identifier error
00000803	I/O error

If portability between CICS and non-CICS environments is required, you can develop a non-EGL program with the same name as the system function (`audit`) to receive the service call in non-CICS environments. When generating for an environment in which the service is not supported, the generator turns the function call into a call to the program with the same name. Alternatively, if the service is not supported in the non-CICS environment, you can test the value of **SysVar.sysType** to determine the runtime environment, then invoke **SysLib.audit()** only if you are running in CICS.

## Behavior in IMS

The record is automatically converted to IMS log format when EGL adds 2 to the length and inserts 2 bytes of binary zeros following the length field. Only the first byte of the record identifier code is used. The second byte of the record identifier code is ignored. The `jid` parameter is ignored. IMS/VS has a maximum limit of 32765 bytes.

## Limitations for support in z/OS batch

To use the **SysLib.audit()** function in z/OS batch, you must specify a PSB for the program, and the program must do at least one of the following:

- References a PSB or PCB in any statement in the program
- Has DL/I databases other than `elawork` or `elamsg` in the PSB definition
- Uses `VGLib.VGTDLI()`
- Associates at least one file with `GSAM`

If these conditions are met, **SysLib.audit()** will behave in the z/OS batch environment just as it does in IMS.

## Example

```
move 32765 to wrkrec.length;  
move "A" to wrkrec.code;  
move 2 to jrnld;  
move "THIS IS THE DATA TO BE WRITTEN TO JOURNAL NUMBER 2" to wrkrec.data;  
SysLib.audit(wrkrec, jrnld);
```

## Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“asynchLink element” on page 459  
“EGL library SysLib” on page 1016  
“errorCode” on page 1068  
“printerAssociation” on page 1059  
“transfer” on page 752

## beginDatabaseTransaction()

The system function **SysLib.beginDatabaseTransaction** begins a relational-database transaction, but only when the EGL runtime is not committing changes automatically. If changes are being committed automatically, the function has no effect.

```
SysLib.beginDatabaseTransaction(  
    [database STRING in])
```

*database*

A database name that was specified in `SysLib.connect` or `VGLib.connectionService`. Use a literal or variable of a character type.

If you do not specify a connection, the function affects the current connection.

When you invoke **SysLib.beginDatabaseTransaction**, a transaction begins at the next I/O operation that uses the specified connection; and the transaction ends when a commit or rollback occurs, as described in *Logical unit of work*. After the commit or rollback, the EGL runtime resumes committing changes automatically.

For details on automatic commits, see *SysLib.connect* and *sqlCommitControl*.

### Related concepts

“Syntax diagram for EGL functions” on page 884  
“Logical unit of work” on page 395  
“SQL support” on page 277

### Related reference

“sqlCommitControl” on page 490  
“connect()” on page 1025  
“connectionService()” on page 1047

## bytes()

The system function **SysLib.bytes** returns the number of bytes in a named area of memory.

```
SysLib.bytes(field fixedFieldOrArray in)  
returns (result INT)
```

*result*

A numeric item that receives the number of bytes in *field*. Two cases are notable:

- If *field* is an array, the function returns the number of bytes in one element
- If *field* is an SQL record, the function returns the number of bytes in the record, including the extra bytes; for details see *SQL record internals*

*field*

An array, item, or record

### Example():

```
result = SysLib.bytes(myItem);
```

## Related concepts

"Syntax diagram for EGL functions" on page 884

## Related reference

"EGL library SysLib" on page 1016

"Primitive types" on page 34

"SQL record internals" on page 876

## calculateChkDigitMod10()

The system function **SysLib.calculateChkDigitMod10** places a modulus-10 check digit in a character item that begins with a series of integers.

```
SysLib.calculateChkDigitMod10(  
  text anyChar inOut,  
  checkLength SMALLINT in,  
  result SMALLINT inOut)
```

### *text*

A character item that begins with a series of integers. The item must include an additional position for the check digit, which goes immediately to the right of the other integers.

### *checkLength*

An item that contains the number of characters that you want to use from the *text* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

### *result*

An item that receives one of two values:

- 0, if the check digit was created
- 1, if the check digit was not created

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **SysLib.calculateChkDigitMod10** in a function-invocation statement.

**Example:** In the following example, *myInput* is an item of type CHAR and contains the value 1734289; *myLength* is an item of type SMALLINT and contains the value 7; and *myResult* is an item of type SMALLINT:

```
SysLib.verifyChkDigitMod10 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-10 check digit, and in all cases the number at the check-digit position is not considered. The algorithm is described in relation to the example values:

1. Multiply the units position of the input number by 2 and multiply every alternate position, moving right to left, by 2:

$$\begin{array}{l} 8 \times 2 = 16 \\ 4 \times 2 = 8 \\ 7 \times 2 = 14 \end{array}$$

2. Add the digits of the products (16814) to the input-number digits (132) that were not multiplied by 2:

$$1 + 6 + 8 + 1 + 4 + 1 + 3 + 2 = 26$$

3. To get the check digit, subtract the sum from the next-highest number ending in 0:

$$30 - 26 = 4$$

If the subtraction yields 10, the check digit is 0.



In this example, the original characters in myInput become these:

1734284

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library SysLib” on page 1016

### calculateChkDigitMod11()

The system function **SysLib.calculateChkDigitMod11** places a modulus-11 check digit in a character item that begins with a series of integers.

```
SysLib.calculateChkDigitMod11(  
    text anyChar inOut,  
    checkLength SMALLINT in,  
    result SMALLINT inOut)
```

#### *text*

A character item that begins with a series of integers. The item must include an additional position for the check digit, which goes immediately to the right of the other integers.

#### *checkLength*

An item that contains the number of characters that you want to use from the *text* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

#### *result*

An item that receives one of two values:

- 0, if the check digit was created
- 1, if the check digit was not created

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **SysLib.calculateChkDigitMod11** in a function-invocation statement.

**Example:** In the following example, myInput is an item of type CHAR and contains the value 56621869; myLength is an item of type SMALLINT and contains the value 8; and myResult is an item of type SMALLINT:

```
SysLib.verifyChkDigitMod (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-11 check digit, and in all cases the number at the check-digit position is not considered. The algorithm is described in relation to the example values:

1. Multiply the digit at the units position of the input number by 2, at the tens position by 3, at the hundreds position by 4, and so on, but let myLength " 1 be the largest number used as a multiplier; and if more digits are in the input number, begin the sequence again using 2 as a multiplier:

```
6 x 2 = 12  
8 x 3 = 24  
1 x 4 = 4  
2 x 5 = 10  
6 x 6 = 36  
6 x 7 = 42  
5 x 2 = 10
```

2. Add the products of the first step and divide the sum by 11:

$$\begin{aligned}
 & (12 + 24 + 4 + 10 + 36 + 42 + 10) / 11 \\
 & = 138 / 11 \\
 & = 12 \text{ remainder } 6
 \end{aligned}$$

3. To get the check digit, subtract the remainder from 11 to get the self-checking digit:

$$11 - 6 = 5$$

If the remainder is 0 or 1, the check digit is 0.

In this example, the original characters in myInput become these:

56621865

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"EGL library SysLib" on page 1016

### callCmd()

The system function **SysLib.callCmd** runs a system command and waits until the command finishes.

```

SysLib.callCmd(
  commandString STRING in
  [, modeString STRING in
])

```

*commandString*

Identifies the operating-system command to invoke.

*modeString*

The *modeString* can be any character or string item. The item can be in either of two modes:

- *form*: in which each character of input becomes available to the program as it is typed, i.e., every key stroke is passed directly to the command specified.
- *line*: in which input is not available until after the newline character key is used, i.e., no information is sent to the command specified until the ENTER key is pressed, and then the entire line typed is sent to the command.

The system command that is being executed must be visible to the running program. For example, if you execute **callCmd**("mySpecialProgram.exe"), the program "mySpecialProgram.exe" must be in a directory pointed to by the environment variable PATH. You may also specify the complete directory location, for example **callCmd**("program files/myWork/mySpecialProgram.exe").

The **SysLib.callCmd** function is supported only in Java environments.

Use the **SysLib.startCmd** function to run a system command that does not wait until the command finishes.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"EGL library SysLib" on page 1016

"startCmd()" on page 1040

## commit()

The system function **SysLib.commit** saves updates that were made to databases and MQSeries message queues since the last commit. A generated Java program or wrapper also saves the updates done by a remote, CICS-based COBOL program (including updates to CICS recoverable files), but only when the call to the remote COBOL program involves a client-controlled unit of work, as described in *luwControl* in *callLink* element.

```
SysLib.commit( )
```

In most cases, EGL performs a single-phase commit that affects each recoverable manager in turn. On CICS for z/OS, however, **SysLib.commit** results in a CICS SYNCPOINT, which performs a two-phase commit that is coordinated across all resource managers.

**SysLib.commit** releases the scan position and the update locks in any file or databases.

When you use **SysLib.commit** with MQ records, the following statements apply:

- Message queue updates are recoverable only if the *Include message in transaction* option is selected in MQ record part.
- Both message **gets** and **adds** are affected by **commit** and **rollback** for recoverable messages. If a **rollback** is issued following a **get** for a recoverable message, the message is placed back on the input queue so that the input message is not lost when the transaction fails to complete successfully. Also, if a **rollback** is issued following an **add** for a recoverable message, the message is deleted from the queue.

You can enhance performance by avoiding unnecessary use of **SysLib.commit**. For details on when an implicit commit occurs, see *Logical unit of work*.

### Example:

```
sysLib.commit();
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

“Logical unit of work” on page 395

“MQSeries support” on page 336

“Run unit” on page 866

“SQL support” on page 277

### Related reference

“commitOnConverse” on page 1057

“segmentedMode” on page 1061

“EGL library SysLib” on page 1016

“luwControl in callLink element” on page 506

“open” on page 722

“prepare” on page 736

## conditionAsInt()

The system function **SysLib.conditionAsInt** accepts a logical expression (like *myVar == 6*), returning a 1 if the expression is true, a 0 if the expression is false.

```
SysLib.conditionAsInt(logicalExpression AnyLogicalExpression in)  
returns (result SMALLINT)
```

*result*

A value of type SMALLINT.

### *logicalExpression*

A logical expression, as described in *Logical expressions*.

### Example:

```
myField = -5;

// result = 0
result = SysLib.conditionAsInt(myField == 6);
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library SysLib” on page 1016

“Logical expressions” on page 593

## connect()

The system function **SysLib.connect** allows a program to connect to a database at run time. This function does not return a value.

```
SysLib.connect(  
  database STRING in,  
  userID STRING in,  
  password STRING in  
  [, commitScope enumerationCommitScope in  
  [, disconnectOption enumerationDisconnectOption in  
  [, isolationLevel enumerationIsolationLevel in  
  [, commitControl enumerationCommitControlOption in  
  ]]] )
```

### *database*

Identifies a database:

- Setting *database* to RESET reconnects to the default database, but if the default database is not available, the connection status remains unchanged; for further details, see *Default database*.
- Otherwise, the physical database name is found by looking up the property **vgj.jdbc.database.server**, where *server* is the name of the database specified on the **SysLib.connect** call. If this property is not defined, the database name that is specified on the **SysLib.connect** call is used as is.
- The format of the database name is different for J2EE connections as compared with non-J2EE connections:
  - If you generated the program for a J2EE environment, use the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB. This situation occurs if build descriptor option **J2EE** was set to YES.
  - If you generated the program for a non-J2EE JDBC environment, use a connection URL; for example, jdbc:db2:MyDB. This situation occurs if option **J2EE** was set to NO.

### *userID*

UserID used to access the database. The argument must be an item of type CHAR and length 8, and a literal is valid. The argument is required. For background information, see *Database authorization and table names*.

### *password*

Password used to access the database. The argument must be an item of type CHAR and length 8, and a literal is valid. The argument is required.

#### *commitScope*

The value is one of the following words, and you cannot use quotes and cannot use a variable:

##### **type1 (the default)**

Only a *one*-phase commit is supported. A new connection closes all cursors, releases locks, and ends any existing connection; nevertheless, invoke **SysLib.commit** or **SysLib.rollback** before making a type1 connection.

If you use type1 as the value of *commitScope*, the value of parameter *disconnectOption* must be the word *explicit*, as is the default.

##### **type2**

A connection to a database does not close cursors, release locks, or end an existing connection. Although you can use multiple connections to read from multiple databases, you should update only one database in a unit of work because only a one-phase commit is available.

##### **twophase**

Identical to type2.

#### *disconnectOption*

This parameter is meaningful only if you are generating Java output. The value is one of the following words, and you cannot use quotes and cannot use a variable:

##### **explicit (the default)**

The connection remains active after the program invokes **SysLib.commit** or **SysLib.rollback**. To release connection resources, a program must issue **SysLib.disconnect**.

If you use type1 as the value of *commitScope*, the value of parameter *disconnectOption* must be set (or allowed to default) to the word *explicit*.

##### **automatic**

A commit or rollback ends an existing connection.

##### **conditional**

A commit or rollback automatically ends an existing connection unless a cursor is open and the hold option is in effect for that cursor. For details on the hold option, see *open*.

#### *isolationLevel*

This parameter indicates the level of independence of one database transaction from another.

The following words are in order of increasing strictness, and as before, you cannot use quotes and cannot use a variable:

- **readUncommitted**
- **readCommitted**
- **repeatableRead**
- **serializableTransaction** (the default value)

For details, see the JDBC documentation from Sun Microsystems, Inc.

#### *commitControl*

This parameter specifies whether a commit occurs after every change to the database.

Valid values are as follows:

- **noAutoCommit** (the default) means that the commit is not automatic, which usually results in faster execution. For details on the rules of commit and rollback in this case, see *Logical unit of work*.
- **autoCommit** means that updates take effect immediately.

You can switch from `autoCommit` to `noAutoCommit` temporarily. For details, see *SysLib.beginDatabaseTransaction*.

**Definition considerations:** `SysLib.connect` sets the following system variables:

- `VGVar.sqlerrd[3]`
- `SysVar.sqlca`
- `SysVar.sqlcode`
- `VGVar.sqlWarn[2]`

**Example:**

```
SysLib.connect(myDatabase, myUserId, myPassword);
```

**Related concepts**

“Logical unit of work” on page 395

“Run unit” on page 866

“SQL support” on page 277

**Related tasks**

“Syntax diagram for EGL statements and commands” on page 884

“Setting up a J2EE JDBC connection” on page 445

“Understanding how a standard JDBC connection is made” on page 309

**Related reference**

“Database authorization and table names” on page 557

“Default database” on page 298

“EGL library SysLib” on page 1016

“Java runtime properties (details)” on page 642

“open” on page 722

“sqlDB” on page 490

“beginDatabaseTransaction()” on page 1020

“disconnect()” on page 1030

“sqlca” on page 1071

“sqlcode” on page 1072

“sqlerrd” on page 1085

“sqlerrmc” on page 1086

“sqlWarn” on page 1087

**convert()**

The system function `SysLib.convert` converts data between EBCDIC (host) and ASCII (workstation) formats or performs code-page conversion within a single format. You can use `SysLib.convert` as the function name in a function invocation statement.

```
SysLib.convert(  
    target anyFixedItemOrRecordOrFormVariable inout,  
    direction enumerationConversionDirection in,  
    conversionTable CHAR(8) in)
```

*target*

Name of the record, data item, or form that has the format you want to

convert. The data is converted in place based on the item definition of the lowest-level items (items with no substructure) in the target object.

Variable-length records are converted only for the length of the current record. The length of the current record is calculated using the record's `numElementsItem` or is set from the record's `lengthItem`. A conversion error occurs and the program ends if the variable-length record ends in the middle of a numeric field or a DBCHAR character.

*direction*

Direction of conversion. "R" and "L" (including the quotation marks) are the only valid values. Required if *conversionTable* is specified; optional otherwise.

"R" Default value. The data is assumed to be in remote format and is converted to local format.

"L" Data is assumed to be in local format and is converted to remote format (as defined in the conversion table).

*conversionTable*

Data item or literal (eight characters, optional) that specifies the name of the conversion table to be used for data conversion. The default value is the conversion table associated with the national language code specified when the program was generated.

**Definition considerations:** You can use the linkage options part to request that automatic data conversion be generated for remote calls, to start remote asynchronous transactions, or for remote file access. Automatic conversion is always performed using the data structure defined for the argument being converted. If an argument has multiple formats, do not request automatic conversion. Instead, code the program to explicitly call **SysLib.convert** with redefined record declarations that correctly map the current values of the argument.

**Example:**

```
Record RecordA
  record_type char(3);
  item1 char(20);
end

Record RecordB
  record_type char(3);
  item2 bigint;
  item3 decimal(7);
  item4 char(8);
end

Program ProgramX type basicProgram
  myRecordA RecordA;
  myRecordB RecordB {redefines = "myRecordA"};
  myConvTable char(8);

  function main();
    myConvTable = "ELACNENU"; // conversion table for US English
    if (myRecordA.record_type == "00A")
      SysLib.convert(myRecordA, "L", myConvTable);
    else;
      SysLib.convert(myRecordB, "L", myConvTable);
    end
    call ProgramY myRecordA;
  end
end
```

## Related concepts

"Syntax diagram for EGL functions" on page 884

## Related reference

"Data conversion" on page 558

"EGL library SysLib" on page 1016

"callConversionTable" on page 1066

## defineDatabaseAlias()

The system function **SysLib.defineDatabaseAlias** creates an alias that can be used to establish a new connection to a database to which your code is already connected. Once established, the alias can be used in any of these functions:

- SysLib.connect
- SysLib.disconnect
- SysLib.beginDatabaseTransaction
- SysLib.setCurrentDatabase
- VGLib.connectionService

The alias can also be used in the **connectionName** field of a variable of type **ReportData**.

```
SysLib.defineDatabaseAlias(  
    alias STRING in,  
    database STRING in)
```

*alias*

A string literal or variable that acts as an alias of the connection identified in the second parameter. The alias is case-insensitive.

*database*

A database name that was specified in SysLib.connect or VGLib.connectionService. Use a literal or variable of a character type.

If you do not specify a connection, the function affects the current connection.

Examples are as follows:

```
// Connect to a database with alias "alias",  
// which becomes the current connection.  
defineDatabaseAlias( "alias", "database" );  
connect( "alias", "user", "pwd" );  
  
// Make two connections to the same database.  
String db = "database";  
defineDatabaseAlias( "alias1", db );  
defineDatabaseAlias( "alias2", db );  
connect( "alias1", "user", "pwd" );  
connect( "alias2", "user", "pwd" );  
  
// Another way to make two connections  
// to the same database.  
defineDatabaseAlias( "alias", "database" );  
connect( "alias", "user", "pwd" );  
connect( "database", "user", "pwd" );  
  
// An alias is defined but not used. The second  
// connect() does not create a new connection.  
defineDatabaseAlias( "alias", "database" );  
connect( "database", "user", "pwd" );  
connect( "database", "user", "pwd" );  
  
// Use of an alias (which is case-insensitive)
```



```
// when disconnecting.
defineDatabaseAlias( "alias", "database" );
connect( "aLiAs", "user", "pwd" );
disconnect( "ALIAS" );

// The next disconnect call fails because the
// connection is called "alias" not "database".
defineDatabaseAlias( "alias", "database" );
connect( "alias", "user", "pwd" );
disconnect( "database" );

// An alias may change. After the next call,
// "alias" refers to "firstDatabase"
defineDatabaseAlias( "alias", "firstDatabase" );

// After the next call,
// "alias" refers to "secondDatabase".
defineDatabaseAlias( "alias", "secondDatabase" );

// The last call would have failed
// if a connection was in place with "alias".
```

### Related concepts

“Syntax diagram for EGL functions” on page 884  
 “SQL support” on page 277

### Related reference

“beginDatabaseTransaction()” on page 1020  
 “connect()” on page 1025  
 “disconnect()”  
 “setCurrentDatabase()” on page 1036  
 “connectionService()” on page 1047

## disconnect()

The system function **SysLib.disconnect** disconnects from the specified database or (if no database is specified) from the current database.

```
SysLib.disconnect(  
    [database STRING in  
    ])
```

*database*

A database name that was specified in SysLib.connect or VGLib.connectionService. Use a literal or variable of a character type.

Before disconnecting, invoke SysLib.commit or SysLib.rollback.

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library SysLib” on page 1016  
 “commit()” on page 1024  
 “connect()” on page 1025  
 “rollback()” on page 1036  
 “connectionService()” on page 1047

## disconnectAll()

The system function **SysLib.disconnectAll** disconnects from all the currently connected databases.

Before disconnecting, invoke **SysLib.commit** or **SysLib.rollback**.

**SysLib.disconnectAll**( )

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library SysLib” on page 1016

“connect()” on page 1025

“connectionService()” on page 1047

### errorLog()

The system function **SysLib.errorLog** copies text into the error log that was started by the system function **SysLib.startLog**.

**SysLib.errorLog**(*text* **STRING** in)

*text*

The value to be placed in the error log.

Log entries include the date and time when the entry was written.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library SysLib” on page 1016

“startLog()” on page 1041

### getCmdLineArg()

The system function **SysLib.getCmdLineArg** returns the specified argument from the list of arguments with which the EGL program was invoked. The specified argument is returned as a string value.

**SysLib.getCmdLineArg**(*index* **INT** in)  
returns (*result* **STRING**)

*result*

The *result* can be any character item.

*index*

The *index* can be any integer item.

- If *index* = 0, the command name is returned.
- If *index* = *n*, the *n*th argument name is returned.
- If *n* is greater than the argument count, a blank is returned.

The following code example loops through the argument list:

```
count int;  
argument char(20);  
  
count = 0;  
argumentCount = SysLib.getCmdLineArgCount();  
  
while (count < argumentCount)  
    argument = SysLib.getCmdLineArg(count)  
    count = count + 1;  
end
```

The **SysLib.getCmdLineArg** function is supported only in Java environments.

Use the **SysLib.getCmdLineArgCount** function to get the number of arguments or parameters that were passed to the main EGL program at the time of its invocation.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library SysLib” on page 1016

“getCmdLineArgCount()”

### getCmdLineArgCount()

The system function **SysLib.getCmdLineArgCount** returns the number of arguments that were used to start the main EGL program.

```
SysLib.getCmdLineArgCount( )  
returns (result INT)
```

*result*

The *result* is the number of arguments.

The following code example loops through the argument list:

```
count int;  
argument char(20);  
  
count = 0;  
argumentCount = SysLib.getCmdLineArgCount();  
  
while (count < argumentCount)  
    argument = SysLib.getCmdLineArg(count)  
    count = count + 1;  
end
```

The **SysLib.getCmdLineArgCount** function is supported only in Java environments.

Use the **SysLib.getCmdLineArg** function to get the specified argument from the list of arguments with which the EGL program was invoked.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library SysLib” on page 1016

“getCmdLineArg()” on page 1031

### getMessage()

The system function **SysLib.getMessage** returns a message from the file that is referenced in the Java runtime property `vgj.message.file`. You can specify inserts for inclusion in the message. After retrieving the message, you can display it in a text form, print form, console form, Web page, or log file.

```
SysLib.getMessage(  
    key STRING in  
    [, insertArray STRING[] in])  
returns (result STRING)
```

*result*

A field of type **STRING**.

*key*

A character field or literal of type `STRING`. This parameter provides the key into the properties file that is used at run time. If the key is blank, the message is a concatenation of message inserts.

*insertArray*

An array of type `STRING`. Each element contains an insert for inclusion in the message being retrieved.

In the message text, the substitution symbol is an integer surrounded by braces, as in this example from a properties file:

```
VGJ0216E = {0} is not a valid date mask for {1}.
```

The first element in *insertArray* is assigned to the placeholder numbered zero, the second is assigned to the placeholder numbered one, and so forth.

The format of the file referenced by Java runtime property `vgj.messages.file` is the same as for any Java properties file. For details on that format, see *Program properties file*.

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"Java runtime properties" on page 431

"Program properties file" on page 433

#### **Related reference**

"EGL library SysLib" on page 1016

"Java runtime properties (details)" on page 642

### **getProperty()**

The system function **SysLib.getProperty** retrieves the value of a Java runtime property. If the specified property is not found, the function returns a null string (`""`).

```
SysLib.getProperty(propertyName STRING in)  
returns (result STRING)
```

*result*

A field of type `STRING`

*propertyName*

A character variable or constant, or a string literal

#### **Related concepts**

"Syntax diagram for EGL functions" on page 884

"Java runtime properties" on page 431

#### **Related reference**

"EGL library SysLib" on page 1016 "Java runtime properties (details)" on page 642

### **loadTable()**

The system function **SysLib.loadTable** loads data from a file into a relational database.

```
SysLib.loadTable(  
  fileName STRING in,  
  insertIntoClause STRING in  
  [, delimiter STRING in  
  ])
```

*fileName*

The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

*insertIntoClause*

Specify the table and columns that will provide the data. Use the syntax of an INSERT clause in an SQL INSERT statement, as in this example:

```
"INSERT INTO myTable(column1, column2)"
```

A clause like the following is sufficient if the file includes values for all table columns in column order:

```
"INSERT INTO myTable"
```

*delimiter*

Specifies the symbol that separates one value from the next in the file. (One row of data must be separated from the next by the newline character.)

The default symbol for *delimiter* is the value in the Java runtime property **vgj.default.databaseDelimiter**; and the default value for that property is a pipe (`|`).

The following symbols are not available:

- Hexadecimal characters (*0* through *9*, *a* through *f*, *A* through *F*)
- Backslash (`\`)
- The newline character or *CONTROL-J*

To unload information from a relational database table and insert it into a file, use the **SysLib.unloadTable** function.

#### Related reference

"EGL library SysLib" on page 1016

"Java runtime properties (details)" on page 642

"unloadTable()" on page 1043

### maximumSize()

The system function **SysLib.maximumSize** returns the maximum number of rows that can be in a dynamic array; specifically, the function returns the value of the array property **maxSize**.

```
SysLib.maximumSize(arrayName anyArray in)
returns (result INT)
```

*result* Maximum number of rows.

*arrayName*

Name of the dynamic array.

**Definition considerations:** The item to which the value is returned must be of type INT or the following equivalent: type BIN with length 9 and no decimal places.

The array name may be qualified by a package name, a library name, or both

An error occurs if you reference an item or record that is not a dynamic array.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"Arrays" on page 75

"EGL library SysLib" on page 1016

## purge()

The system function **SysLib.purge** deletes a CICS temporary storage queue.

**SysLib.purge**(*queueName* **STRING** in)

*queueName*

Either a literal or an item of type CHAR (1 to 8 bytes). *queueName* is required and contains the name of a single temporary storage queue.

If you use **SysLib.purge** as the function name in a function invocation statement, the program runs the CICS ENQ command with the **NOSUSPEND** option to enqueue on the resource name **EZETEMP-*queuename***. The program runs the CICS DEQ command to dequeue after the temporary storage queue is deleted. If an error occurs, the first byte of the **EIBFN** is placed in the first 2 characters of **SysVar.errorCode**, and bytes 0 to 2 of the **EIBRCODE** are placed in the last 6 characters of **SysVar.errorCode**.

**Example:** The following deletes the CICS temporary storage queue associated with the current value of **resourceAssociation** for record ABC (where *myqueue* is an item name):

```
myqueue = abc.resourceAssociation;  
SysLib.purge(myqueue);
```

The following deletes the CICS temporary storage queue associated with destination XYZ:

```
SysLib.purge("XYZ");
```

### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"EGL library SysLib" on page 1016

## queryCurrentDatabase()

The system function **SysLib.queryCurrentDatabase** returns the product and release number of the currently connected database.

**SysLib.queryCurrentDatabase**(  
  *product* **CHAR(8)** inOut,  
  *release* **CHAR(8)** inOut)

*product*

Receives the database product name. The argument must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

*release*

Receives the database release level. The argument must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library SysLib” on page 1016

### rollback()

The system function **SysLib.rollback** reverses updates that were made to databases and MQSeries message queues since the last commit. That reversal occurs in any EGL-generated application.

**SysLib.rollback( )**

A rollback occurs automatically when a program ends as a result of an error condition.

**Definition considerations:** When you use **SysLib.rollback** with MQ records, the following statements apply:

- Message queue updates are recoverable only if the *Include message in transaction* option is selected in MQ record part.
- Both message **scans** and **adds** are affected by **commit** and **rollback** for recoverable messages. If a **rollback** is issued following a **scan** for a recoverable message, the message is placed back on the input queue so that the input message is not lost when the transaction fails to complete successfully. Also, if a **rollback** is issued following an **add** for a recoverable message, the message is deleted from the queue.

#### Target platforms:

Platform	Compatibility considerations
iSeries, USS, Windows 2000, Windows NT	Reverses changes to relational databases and MQSeries message queues, as well as changes made to remote server programs that were called using client-controlled unit of work.

#### Example:

```
SysLib.rollback();
```

#### Related concepts

“Syntax diagram for EGL functions” on page 884

“Logical unit of work” on page 395

“MQSeries support” on page 336

“SQL support” on page 277

#### Related reference

“EGL library SysLib” on page 1016

### setCurrentDatabase()

The system function **SysLib.setCurrentDatabase** makes the specified database the currently active one.

**SysLib.setCurrentDatabase**(*database* **STRING** in)

*database*

A database name that was specified in SysLib.connect or VGLib.connectionService. Use a literal or variable of a character type.

#### Related concepts

“Syntax diagram for EGL functions” on page 884

#### Related reference

“EGL library SysLib” on page 1016

“connect()” on page 1025

“connectionService()” on page 1047

### setError()

The system function **SysLib.setError** associates a message with a field in a PageHandler or VGUIRecord or with a PageHandler as a whole:

- If a PageHandler invokes the function, the message is placed at the location of a message or messages tag in the JSP.
- If a program of type VGWebTransaction invokes the function, the message is placed at the location of the font tag in the JSP.

The message is displayed when the related Web page is displayed.

If a validation function invokes **SysLib.setError**, the Web page is re-displayed automatically when the function ends.

The first of the following three syntaxes are available in PageHandlers or in programs of type VGWebTransaction, while the second and third are available only for PageHandlers:

```
SysLib.setError(  
    fieldInError anyPageField in,  
    msgKey STRING in  
    {, fieldInsert sysLibFieldInsert in})  
  
SysLib.setError(  
    this enumerationThis in,  
    msgKey STRING in  
    {, fieldInsert sysLibFieldInsert in})  
  
SysLib.setError(msgText STRING in)
```

*fieldInError*

If **SysLib.setError** is issued from a PageHandler, *fieldInError* is a field in the PageHandler.

If **SysLib.setError** is issued from a program, *fieldInError* is a field in the record that is specified in the **show** or **converse** statement.

**this**

Refers to the PageHandler from which **SysLib.setError** is issued. In this case, the message is not specific to a field, but is associated with the PageHandler as a whole. For details on **this**, see *References to variables and constants*.

*msgKey*

A character field or literal (type CHAR or MBCHAR) that provides the key into the message resource bundle or properties file used at run time. If the key is blank, the message is a concatenation of message inserts.



### *fieldInsert*

The character field or literal that is included as an insert to the output message. The substitution symbol in message text is an integer surrounded by braces, as in this example:

Invalid file name {0}

### *msgText*

The character field or literal that you can specify if you do not specify other arguments. The text is associated with the page as a whole.

You can associate multiple messages with a field or with the PageHandler. The messages are lost if control is forwarded; specifically, if the PageHandler runs a **forward** statement or if the VGWebTransaction program runs a **transfer** statement.

### **Related concepts**

“PageHandler” on page 223

“References to variables in EGL” on page 59

### **Related tasks**

“Syntax diagram for EGL statements and commands” on page 884

### **Related reference**

“EGL library SysLib” on page 1016

“forward” on page 686

## **setLocale()**

The system function **SysLib.setLocale** is used in PageHandlers and in programs of type VGWebTransaction. The function sets the Java locale, which determines these aspects of runtime behavior:

- The human language used for labels and messages
- The default date and time formats

You might present a list of languages on a Web page, for example, and set the Java locale based on the user’s selection. The new Java locale is in use until one of the following occurs:

- You invoke **SysLib.setLocale** again; or
- The browser session ends; or
- If the Web page was presented as a result of your issuing a **converse** statement from the current program of type VGWebTransaction, the user submits a form or clicks a link that does not re-invoke that program; or
- If the Web page was presented as a result of your issuing a **show** statement with a returning clause, the user submits a form or clicks a link that does not re-invoke the program that the statement indicated would be invoked; or
- A new Web page is presented otherwise.

In the cases mentioned, the next Web page reverts (by default) to the Java locale specified in the browser.

If the user submits a form or clicks a link that opens a new window, the Java locale in the original window is unaffected by the locale in the new window.

**SysLib.setLocale** conforms to the JDK 1.1 and 1.2 API documentation for class `java.util.Locale`. See ISO 639 for language codes and ISO 3166 for country codes.

```
SysLib.setLocale(  
    languageCode CHAR(2) in,  
    countryCode CHAR(2) in  
    [, variant CHAR(2) in])
```

*languageCode*

A two-character language code specified as a literal or contained in an item of type CHAR. Only language codes that are defined by ISO 639 are valid.

*countryCode*

A two-character country code specified as a literal or contained in an item of type CHAR. Only country codes that are defined by ISO 3166 are valid.

*variant*

A variant, which is a code specified as a literal or contained in an item of type CHAR. This code is not part of a Java specification but depends on the browser and other aspects of the user environment.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

“PageHandler” on page 223

#### **Related reference**

“EGL library SysLib” on page 1016

### **setRemoteUser()**

The system function **SysLib.setRemoteUser** sets the userid and password that are used on calls to remote programs.

```
SysLib.setRemoteUser(  
    userID STRING in,  
    password STRING in)
```

*userID*

The user ID on the remote system.

*password*

The password on the remote system.

When the linkage options part, callLink element, property remoteComType is CICSJ2C, CICSECI, or JAVA400 on a remote call, authorization is based on the values (if non-blank) that are passed to **SysLib.setRemoteUser**. If a value is blank or not specified, the value is sought in the file **csouidpwd.properties**, which includes the properties CSOUID (for the user ID) and CSOPWD (for the password). If you use neither approach, EGL runtime makes the call without a username and password.

Before invoking **SysLib.setRemoteUser**, your code can issue Java access functions that display a dialog box to prompt the user for the user ID and password. You can use one or both values in **csouidpwd.properties** as a default that takes effect when the user does not provide the information.

#### **Related concepts**

“Syntax diagram for EGL functions” on page 884

#### **Related reference**

“csouidpwd.properties file for remote calls” on page 460

“EGL library SysLib” on page 1016

“remoteComType in callLink element” on page 511

## size()

The system function **SysLib.size** returns the number of rows in the specified data table or the number of elements in the specified array. The array may be a structure-item array or a dynamic array of data items or records.

**SysLib.size**(*arrayName* **anyArray** *in*)  
returns (*result* **INT**)

*result*

The number of rows in the specified data table or the number of elements in the specified array.

*arrayName*

Name of the array or data table.

**Definition considerations:** The item to which the value is returned must be of type INT or the following equivalent: type BIN with length 9 and no decimal places.

If the array name (*arrayName*) is in a substructured element of another array, the returned value is the number of occurrences for the structure item itself, not the total number of occurrences in the containing structure (see *Examples* section).

The array name may be qualified by a package name, a library name, or both

An error occurs if you reference an item or record that is not an array.

**Examples:** This example uses the value returned by **SysLib.size** to control a loop:

```
// Calculate the sum of an array of numbers
sum = 0;
i = 1;
myArraySize = SysLib.size(myArray);

while (i <= myArraySize)
    sum = myArray[i] + sum;
    i = i + 1;
end
```

Next, consider the following record part:

```
Record myRecordPart
    10 siTop CHAR(40)[3];
    20 siNext CHAR(20)[2];
end
```

Given that you create a record based on myRecordPart, you can use **SysLib.size(siNext)** to determine the occurs value for the subordinate array:

```
// Sets count to 2
count = SysLib.size(myRecord.siTop.siNext);
```

## Related concepts

“Syntax diagram for EGL functions” on page 884

## Related reference

“Arrays” on page 75

“EGL library SysLib” on page 1016

## startCmd()

The system function **SysLib.startCmd** runs a system command and does not wait until the command finishes.

```
SysLib.startCmd(
    commandString STRING in
    [, modeString STRING in
    ])
```

*commandString*

Identifies the operating-system command to invoke.

*modeString*

The *modeString* can be any character or string item. The item can be in either of two modes:

- *form*: in which each character of input becomes available to the program as it is typed, i.e., every key stroke is passed directly to the command specified.
- *line*: in which input is not available until after the newline character is used, i.e., no information is sent to the command specified until the ENTER key is pressed, and then the entire line typed is sent to the command.

The system command that is being executed must be visible to the running program. For example, if you execute **callCmd**("mySpecialProgram.exe"), the program "mySpecialProgram.exe" must be in a directory pointed to by the environment variable PATH. You may also specify the complete directory location, for example **callCmd**("program files/myWork/mySpecialProgram.exe").

The **SysLib.startCmd** function is supported only in Java environments.

Use the **SysLib.callCmd** function to run a system command which waits until the command finishes.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"EGL library SysLib" on page 1016

### startLog()

The system function **SysLib.startLog** opens an error log. Text is written into that log every time your program invokes **SysLib.errorLog**.

```
SysLib.startLog(logFile STRING in)
```

*logFile*

The error log.

#### Related concepts

"Syntax diagram for EGL functions" on page 884

#### Related reference

"EGL library SysLib" on page 1016

"errorLog()" on page 1031

### startTransaction()

The system function **SysLib.startTransaction()** invokes a main program asynchronously, associates that program with a printer or terminal device, and passes a record. If the receiving program is generated by EGL, the record is used to initialize the input record; if the receiver is produced by VisualAge Generator, the record is used to initialize the working storage.

The default behavior of this function is to start a program that resides in the same Java package. To change that behavior on CICS, specify an `asynchLink` element in the linkage options part that is used to generate the invoking program.

A Java program can transfer only to another Java program on the same machine.

```
SysLib.startTransaction(  
  request anyBasicRecord in  
  [, prID startTransactionPrId in  
  [, termID CHAR(4) in ]])
```

*request*

The name of a basic record, which must have the following format:

- The first 2 bytes (of type `SMALLINT` or of type `BIN` without decimals) contain the length of the data to be passed to the started transaction, plus 10 for the two fields (including this one) that are not passed.
- The next 8 bytes (of type `CHAR`) are also not passed, but contain the name of the program to be started.
- The remaining part of the request record is passed.

*prID*

This 4-byte item is ignored if specified.

*termID*

This 4-byte item of type `CHAR` is ignored if specified. You must specify *prID* if you specify *termID*.

## IMS considerations

**SysLib.startTransaction()** results in an insert to the modifiable alternate PCB. The indicated work area is passed as the message. The generated COBOL program automatically adds an extra 2 bytes between the length and the transaction and adds 2 to the length value. The transaction is started without an associated terminal. `Prid` and `recip` are ignored.

The maximum length on the request record is 32765 bytes.

The transaction that is started must be included in the system `IMSGEN` and must be defined as a nonconversational transaction. If the started transaction is not an EGL program, it must issue a GU call to the I/O PCB to retrieve the message.

**SysLib.startTransaction()** cannot start a program at a remote system.

If the call was nested in a **try** block, `sysVar.errorCode` can contain the value 00000203, indicating the call failed, probably because of an invalid transaction identifier.

### Related concepts

"Syntax diagram for EGL functions" on page 884

### Related reference

"asynchLink element" on page 459

"EGL library SysLib" on page 1016

"errorCode" on page 1068

"printerAssociation" on page 1059

"transfer" on page 752

## unloadTable()

The system function **SysLib.unloadTable** unloads data from a relational database into a file.

```
SysLib.unloadTable(  
    fileName STRING in,  
    selectStatement STRING in  
    [, delimiter STRING in       
    ])
```

*fileName*

The name of the file. The name is fully qualified or is relative to the directory from which the program is invoked.

*selectStatement*

Specify the criteria for selecting data from the relational database. Use the syntax of an SQL SELECT statement without including host variables; for example:

```
"SELECT column1, column2 FROM myTABLE  
WHERE column3 > 10"
```

*delimiter*

Specifies the symbol that will separate one value from the next in the file. (One row of data must be separated from the next by the newline character.)

The default symbol for *delimiter* is the value in the Java runtime property **vgj.default.databaseDelimiter**; and the default value for that property is a pipe (`|`).

The following symbols are not available:

- Hexadecimal characters (*0* through *9*, *a* through *f*, *A* through *F*)
- Backslash (`\`)
- The newline character or *CONTROL-J*

To load information from a file and insert it into a relational database table, use the **SysLib.loadTable** function.

### Related reference

"EGL library SysLib" on page 1016

"loadTable()" on page 1033

"Java runtime properties (details)" on page 642

## verifyChkDigitMod10()

The system function **SysLib.verifyChkDigitMod10** verifies a modulus-10 check digit in a character item that begins with a series of integers.

```
SysLib.verifyChkDigitMod10(  
    text anyChar in,  
    checkLength SMALLINT in,  
    result SMALLINT inOut)
```

*text*

A character item that begins with a series of integers. The item include an additional position for the check digit, which is immediately to the right of the other integers.

*checkLength*

An item that contains the number of characters that you want to use from the *text* item, including the position used for the check digit. This item has 4 digits and is either of type **SMALLINT** or is of a type **BIN**, with no decimal places.

*result*

An item that receives one of two values:

- 0, if the calculated check digit matches the value in *text*
- 1, if the calculated check digit does not match that value

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **SysLib.verifyChkDigitMod10** in a function-invocation statement; or as an item validator in a text form.

**Example:** In the following example, myInput is an item of type CHAR and contains the value 1734284; myLength is an item of type SMALLINT and contains the value 7; and myResult is an item of type SMALLINT:

```
SysLib.verifyChkDigitMod10 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-10 check digit, and in all cases the number at the check-digit position is not considered; but when the algorithm is complete, the calculated value is compared with the number at the check-digit position.

The algorithm is described in relation to the example values:

1. Multiply the units position of the input number by 2 and multiply every alternate position, moving right to left, by 2:

$$8 \times 2 = 16$$

$$4 \times 2 = 8$$

$$7 \times 2 = 14$$

2. Add the digits of the products (16814) to the input-number digits (132) that were not multiplied by 2:

$$1 + 6 + 8 + 1 + 4 + 1 + 3 + 2 = 26$$

3. To get the check digit, subtract the sum from the next-highest number ending in 0:

$$30 - 26 = 4$$

If the subtraction yields 10, the check digit is 0.

In this example, the calculated check digit matches the value in the check-digit position, and the value of myResult is 0.

#### Related reference

“EGL library SysLib” on page 1016

“Validation properties” on page 68

### verifyChkDigitMod11()

The system function **SysLib.verifyChkDigitMod11** verifies a modulus-11 check digit in a character item that begins with a series of integers.

```
SysLib.verifyChkDigitMod11(  
  text anyChar in,  
  checkLength SMALLINT in,  
  result SMALLINT inOut)
```

*text*

A character item that begins with a series of integers. The item include an additional position for the check digit, which is immediately to the right of the other integers.

### *checkLength*

An item that contains the number of characters that you want to use from the *text* item, including the position used for the check digit. This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

### *result*

An item that receives one of two values:

- 0, if the calculated check digit matches the value in *text*
- 1, if the calculated check digit does not match that value

This item has 4 digits and is either of type SMALLINT or is of a type BIN, with no decimal places.

You can use **SysLib.verifyChkDigitMod11** in a function-invocation statement; or as an item validator in a text form.

**Example:** In the following example, *myInput* is an item of type CHAR and contains the value 56621869; *myLength* is an item of type SMALLINT and contains the value 8; and *myResult* is an item of type SMALLINT:

```
sysLib.verifyChkDigitMod11 (myInput, myLength, myResult);
```

An algorithm is used to derive the modulus-11 check digit, and in all cases the number at the check-digit position is not considered; but when the algorithm is complete, the calculated value is compared with the number at the check-digit position. The algorithm is described in relation to the example values:

1. Multiply the digit at the units position of the input number by 2, at the tens position by 3, at the hundreds position by 4, and so on, but let *myLength* " 1 be the largest number used as a multiplier; and if more digits are in the input number, begin the sequence again using 2 as a multiplier:

```
6 x 2 = 12
8 x 3 = 24
1 x 4 = 4
2 x 5 = 10
6 x 6 = 36
6 x 7 = 42
5 x 2 = 10
```

2. Add the products of the first step and divide the sum by 11:

```
(12 + 24 + 4 + 10 + 36 + 42 + 10) / 11
= 138 / 11
= 12 remainder 6
```

3. To get the check digit, subtract the remainder from 11 to get the self-checking digit:

```
11 - 6 = 5
```

If the remainder is 0 or 1, the check digit is 0.

In this example, the calculated check digit matches the value in the check-digit position, and the value of *myResult* is 0.

### **Related reference**

"EGL library SysLib" on page 1016

"Validation properties" on page 68

## **wait()**

The system function **SysLib.wait** suspends execution for the specified number of seconds.



**SysLib.wait**(*timeInSeconds* **BIN(9,2)** in)

*timeInSeconds*

The time can be any numeric item or literal. Fractions of a second down to hundredths of seconds are honored if the number is not an integer.

You can use **SysLib.wait** when two asynchronously running programs need to communicate through a record in a shared file or database. One program might need to suspend processing until the other program updates the information in the shared record.

## Example

```
SysLib.wait(15); // waits for 15 seconds
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library SysLib” on page 1016

## writeStderr()

The system function **SysLib.writeStderr()** writes a text string to the stderr destination.

**SysLib.writeStderr**(*textString* **STRING** in)

*textString*

The string to be displayed.

**SysLib.writeStderr()** is used mostly for debugging. The comparable **ConsoleLib.displayLineMode()** is not supported in page handler programs, while **SysLib.writeStderr()** is.

## Example

```
SysLib.writeStderr("Opened database");
```

### Related concepts

“Syntax diagram for EGL functions” on page 884

### Related reference

“EGL library SysLib” on page 1016

“writeStdout()”

## writeStdout()

The system function **SysLib.writeStdout()** writes a text string to the stdout destination.

**SysLib.writeStdout**(*textString* **STRING** in)

*textString*

The string to be displayed.

**SysLib.writeStdout()** is mostly used for debugging. The comparable **ConsoleLib.displayLineMode()** is not supported in page handler programs, while **SysLib.writeStdout()** is.

## Example

```
SysLib.writeStdout("Opened database");
```

## Related concepts

“Syntax diagram for EGL functions” on page 884

## Related reference

“EGL library SysLib” on page 1016

“writeStderr()” on page 1046

## EGL library VGLib

The VGLib functions are shown below:

System function/Invocation	Description
<i>result</i> = compareBytes ( <i>target</i> , <i>targetSubIndex</i> , <i>targetSubLength</i> , <i>source</i> , <i>sourceSubIndex</i> , <i>sourceSubLength</i> )	Compares two values and returns a value (-1, 0, or 1) to indicate which of the two is greater
<i>result</i> = concatenateBytes ( <i>target</i> , <i>source</i>	Concatenates <i>target</i> and <i>source</i> ; places the new value in <i>target</i> ; and returns an integer that indicates whether <i>target</i> was long enough to contain the new value
“connectionService()”( <i>userID</i> , <i>password</i> , <i>serverName</i> [, <i>product</i> , <i>release</i> [, <i>connectionOption</i> ]])	Provides two benefits: <ul style="list-style-type: none"><li>• Allows a program to connect or disconnect to a database at run time.</li><li>• Receives (optionally) the database product name and release level. You can use the received information in a <b>case</b>, <b>if</b>, or <b>while</b> statement so that runtime processing is dependent on characteristics of the database.</li></ul>
copyBytes ( <i>target</i> , <i>targetSubIndex</i> , <i>targetSubLength</i> , <i>source</i> , <i>sourceSubIndex</i> , <i>sourceSubLength</i> )	Copies one value to another
<i>result</i> = “getVAGSysType()” on page 1054()	Identifies the target system in which the program is running.
“EGLTDLI()” on page 930 ( <i>func</i> , <i>pcbindex</i> , <i>parms</i> ...)	Invokes a DL/I function directly using the CBLTDLI interface.

## Related reference

“EGL library DLILib” on page 929

“AIBTDLI()” on page 930

“EGLTDLI()” on page 930

## connectionService()

The system function **VGLib.connectionService** provides two benefits:

- Allows a program to connect or disconnect to a database at run time.
- Receives (optionally) the database product name and release level. You can use the received information in a **case**, **if**, or **while** statement so that runtime processing is dependent on characteristics of the database.

When you use **VGLib.connectionService** to create a new connection from a Java program, specify the isolation level by setting the system variable **VGVar.sqlIsolationLevel**.

**VGLib.connectionService** is for use only in programs migrated from VisualAge Generator and EGL 5.0. The function is supported (at development time) if the EGL preference **VisualAge Generator compatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

For new programs, use these system functions instead:

- SysLib.connect
- SysLib.disconnect
- SysLib.disconnectAll
- SysLib.queryCurrentDatabase
- SysLib.setCurrentDatabase

**VGLib.connectionService** does not return a value.

```
VGLib.connectionService(  
  userID CHAR(8) in,  
  password CHAR(8) in,  
  serverName CHAR(18) in,  
  [, product CHAR(8) inOut,  
    release CHAR(8) inOut  
  [, connectionOption STRING in  
  ]])
```

*userID*

UserID used to access the database. The argument must be an item of type CHAR and length 8; a literal is not valid. The argument is required. For background information, see *Database authorization and table names*.

*password*

Password used to access the database. The argument must be an item of type CHAR and length 8; a literal is not valid. The argument is required.

*serverName*

Specifies a connection and uses that connection to assign values to the arguments *product* and *release*, if those arguments are included in the invocation of **VGLib.connectionService**.

The argument *serverName* is required and must be an item of type CHAR and length 18. Any of the following values are valid:

#### **blanks (no content)**

If a connection is in place, **VGLib.connectionService** maintains that connection. If a connection is not in place, the result (other than to assign values) is to return to the connection status that is in effect at the beginning of a run unit, as described in *Default database*.

#### **RESET**

RESET reconnects to the default database; but if the default database is not available, the connection status remains unchanged.

For further details, see *Default database*.

*serverName*

Identifies a database:

- The physical database name is found by looking up the property **vgj.jdbc.database.server**, where *server* is the name of the server specified on the **VGLib.connectionService** call. If this property is not defined, the server name that is specified on the **VGLib.connectionService** call is used as is.

- The format of the database name is different for J2EE connections as compared with non-J2EE connections:
  - If you generated the program for a J2EE environment, use the name to which the datasource is bound in the JNDI registry; for example, jdbc/MyDB. This situation occurs if build descriptor option **J2EE** was set to YES.
  - If you generated the program for a non-J2EE JDBC environment, use a connection URL; for example, jdbc:db2:MyDB. This situation occurs if option **J2EE** was set to NO.

#### *product*

Receives the database product name. The argument, if any, must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

#### *release*

Receives the database release level. The argument, if any, must be an item of type CHAR and length 8.

To determine the string that will be received when your code connects to a particular database, review the product documentation for the database or driver; or run your code in a test environment and write the received value to a file.

#### *connectionOption*

Valid values are as follows:

##### **D1E**

D1E is the default. The *1* in the option name indicates that only a *one*-phase commit is supported, and the *E* indicates that any disconnect must be *explicit*. In this case, a commit or rollback has no effect on an existing connection.

A connection to a database does not close cursors, release locks, or end an existing connection. If the run unit is already connected to the same database, however, the effect is equivalent to specifying DISC then D1E.

You can use multiple connections to read from multiple databases, but you should update only one database in a unit of work because only a one-phase commit is available.

##### **D1A**

The *1* in the option name indicates that only a *one*-phase commit is supported, and the *A* indicates that any disconnect is *automatic*. Characteristics of this option are as follows:

- You can connect to only one database at a time
- A commit, rollback, or connection to a database ends an existing connection

##### **DISC**

Disconnect from the specified database. Disconnecting from a database causes a rollback and releases locks, but only for that database.

##### **DCURRENT**

Disconnect from the currently connected database. Disconnecting from a database causes a rollback and releases locks, but only for that database.

## DALL

Disconnect from all connected databases. Disconnecting from all databases causes a rollback in those database, but not in other recoverable resources.

## SET

Set a connection current. (By default, the connection most recently made in the run unit is current.)

The following values are supported for compatibility with VisualAge Generator, but are equivalent to D1E: R, D1C, D2A, D2C, D2E.

**Definition considerations:** **VGLib.connectionService** sets the following system variables:

- VGLib.sqlerrd
- SysVar.sqlca
- SysVar.sqlcode
- VGLib.sqlWarn

## Example:

```
VGLib.connectionService(myUserId, myPassword,  
myServerName, myProduct, myRelease, "D1E");
```

## Related concepts

“Syntax diagram for EGL functions” on page 884  
“Logical unit of work” on page 395  
“Run unit” on page 866  
“SQL support” on page 277

## Related tasks

“Syntax diagram for EGL statements and commands” on page 884  
“Setting up a J2EE JDBC connection” on page 445  
“Understanding how a standard JDBC connection is made” on page 309

## Related reference

“Database authorization and table names” on page 557  
“Default database” on page 298  
“EGL library VGLib” on page 1047

“Java runtime properties (details)” on page 642  
“sqlDB” on page 490  
“sqlca” on page 1071  
“sqlcode” on page 1072  
“sqlerrd” on page 1085  
“sqlerrmc” on page 1086  
“sqlIsolationLevel” on page 1086  
“sqlWarn” on page 1087

## compareBytes()

The system function **VGLib.compareBytes** compares two fields, byte by byte.

As is true in **StrLib.compareStr**, the source and target may be of a character type. In the current function, the source also may be any of the following types:

- BIN (or the integer equivalents BIGINT, INT, and SMALLINT)
- DECIMAL
- NUM

- NUMC
- PACF

```
StrLib.compareBytes(
    target a character type in,
    targetIndex INT in,
    targetSubLength INT in,
    source BIN, DECIMAL, NUM, NUMC, PACF, or a character type in,
    sourceIndex INT in,
    sourceSubLength INT in )
returns (result INT)
```

*result*

Numeric field that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

- 1      The value based on *target* is less than the value based on *source*
- 0        The value based on *target* is equal to the value based on *source*
- 1        The value based on *target* is greater than the value based on *source*

*target*

Value from which a target value is derived. Can be a field or a literal.

*targetSubIndex*

Identifies the starting byte of a value in *target*, given that the first byte in *target* has the index value 1. This index can be an integer literal. Alternatively, this index can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*targetSubLength*

Identifies the number of bytes in the value that is derived from *target*. The length can be an integer literal. Alternatively, the field can be defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*source*

Field or literal from which a source value is derived.

*sourceSubIndex*

Identifies the starting byte of the value in *source*, given that the first byte in *source* has the index value of 1. This index can be an integer literal. Alternatively, this index can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*sourceSubLength*

Identifies the number of bytes in the value that is derived from *source*. The length can be an integer literal. Alternatively, this index can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

A byte-to-byte comparison of the values is performed. If the values are not the same length, the shorter value is padded with blanks even that value is numeric.

**Definition considerations:** The following values are returned in **sysVar.errorCode**:

- 8        Index less than 1 or greater than value length.
- 12      Length less than 1.

- 20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character
- 24 Invalid double-byte length. Length in bytes for a DBCHAR or UNICODE string is odd (double-byte lengths must always be even).

**Example:**

```
target = "123456";
source = "34";
result =
  StrLib.compareBytes(target,3,2,source,1,2);
// result = 0
```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“compareStr()” on page 1000

“EGL library VGLib” on page 1047

**concatenateBytes()**

The system function **VGLib.concatenateBytes** concatenates two fields.

As is true in **StrLib.concatenate**, the source and target may be of a character type. In the current function, the source also may be any of the following types:

- BIN (or the integer equivalents BIGINT, INT, and SMALLINT)
- DECIMAL
- NUM
- NUMC
- PACF

```
StrLib.concatenateBytes(
  target a character type inOut,
  source BIN, DECIMAL, NUM, NUMC, PACF, or a character type in)
returns (result INT)
```

*result*

Numeric field that receives one of the following values (defined as type INT or the equivalent: type BIN with length 9 and no decimal places) returned by the function:

- 1 Concatenated output is too long to fit in the target field and the output was truncated, as described later
- 0 Concatenated output fits in the target field

*target*

Target field.

*source*

Source field or literal.

When two values are concatenated, the following occurs:

1. Any trailing spaces or nulls are deleted from the target value
2. The source value is appended to the value produced by the previous step
3. If the output produced by the second step is longer than the target field, the output is truncated. If the output is shorter than the target field, the output is padded with blanks, even if the output is a number

**Example:**

```

phrase = "and/ "; // CHAR(7)
or      = "or";
result =
  VGLib.concatenateBytes(phrase,or);
if (result == 0)
  print phrase; // phrase = "and/or "
end

```

**Related concepts**

“Syntax diagram for EGL functions” on page 884

**Related reference**

“concatenate()” on page 1001

“EGL library VGLib” on page 1047

**copyBytes()**

The system function **VGLib.copyBytes** copies one field to another.

As is true in **StrLib.copyStr**, the source and target may be of a character type. In the current function, the source also may be any of the following types:

- BIN (or the integer equivalents BIGINT, INT, and SMALLINT)
- DECIMAL
- NUM
- NUMC
- PACF

```

StrLib.copyBytes(
  target a character type inOut,
  targetSubIndex INT in,
  targetSubLength INT in,
  source BIN, DECIMAL, NUM, NUMC, PACF, or a character type in,
  sourceSubIndex INT in,
  sourceSubLength INT in)

```

*target*

Field or literal from which a target substring is derived.

*targetSubIndex*

Identifies the starting byte in *target*, given that the first byte in *target* has the value 1. This index can be an integer literal. Alternatively, this index can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*targetSubLength*

Identifies the number of bytes in the value that is derived from *target*. The length can be an integer literal. Alternatively, the length can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

*source*

Field or literal from which a source value is derived.

*sourceSubIndex*

Identifies the starting byte of the value in *source*, given that the first byte in *source* has the value 1. This index can be an integer literal. Alternatively, this index can be a field defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.



### *sourceSubLength*

Identifies the number of bytes in the value that is derived from *source*. The length can be an integer literal. Alternatively, the length can be an item defined as type INT or the following equivalent: type BIN with length 9 and no decimal places.

If the source is longer than the target, the source is truncated. If the source is shorter than the target, the source value is padded with blanks, even if that value is numeric.

**Definition considerations:** The following values are returned in **sysVar.errorCode**:

- 8 Index less than 1 or greater than value length.
- 12 Length less than 1.
- 20 Invalid double-byte index. Index for a DBCHAR or UNICODE string points to middle of double-byte character.
- 24 Invalid double-byte length. Length in bytes for a DBCS or UNICODE string is odd (double-byte lengths must always be even).

### **Example:**

```
target = "120056";
source = "34";
StrLib.copyBytes(target,3,2,source,1,2);
// target = "123456"
```

### **Related concepts**

"Syntax diagram for EGL functions" on page 884

### **Related reference**

"copyStr()" on page 1003

"EGL library VGLib" on page 1047

## **getVAGSysType()**

The system function **VGLib.getVAGSysType** identifies the target system in which the program is running. The function is supported (at development time) if the program property **VAGCompatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

If the generated output is a Java wrapper, **VGLib.getVAGSysType** is not available. Otherwise, the function returns the character value that would have been returned by the VisualAge Generator EZESYS special function word. If the current system was not supported by VisualAge Generator, the function returns the uppercase, string equivalent of the code returned by **SysVar.systemType**.

```
VGLib.getVAGSysType( )
returns (result CHAR(8))
```

### *result*

A character string that contains the system type code, as shown in the next table.

**VGLib.getVAGSysType** returns the VisualAge Generator equivalent of the value in **SysVar.systemType**.

Value in sysVar.systemType	Value returned by VGLib.getVAGSysType
AIX	"AIX"

Value in <code>sysVar.systemType</code>	Value returned by <code>VGLib.getVAGSysType</code>
DEBUG	"ITF"
ISERIESC	"OS400"
ISERIESJ	"OS400"
LINUX	"LINUX"
USS	"OS390"
WIN	"WINNT"

The value returned by **VGLib.getVAGSysType** can be used only as a character string; you cannot use the returned value with the operands *is* or *not* in a logical expression, as you can with **sysVar.systemType**:

```
// valid ONLY for sysVar.systemType
if sysVar.systemType is AIX
    call myProgram;
end
```

The only place that **VGLib.getVAGSysType** can be used is as the source in an assignment or **move** statement.

The characteristics of **VGLib.getVAGSysType** are as follows:

**Primitive type**

CHAR

**Data length**

8 (padded with blanks)

**Is value always restored after a converse?**

Yes

It is recommended that you use **sysVar.systemType** instead of **VGLib.getVAGSysType**.

**Definition considerations:** The value of **VGLib.getVAGSysType** does not affect what code is validated at generation time. For example, the following **add** statement is validated even if you are generating for Windows:

```
mySystem CHAR(8);
mySystem = VGLib.getVAGSysType();
if (mySystem == "AIX")
    add myRecord;
end
```

To avoid validating code that will never run in the target system, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

```
mySystem CHAR(8);
mySystem = VGLib.getVAGSysType();

if (mySystem == "AIX")
    call myAddProgram myRecord;
end
```

An alternative way to solve the problem is available, but only if you use **sysVar.systemType** instead of **VGLib.getVAGSysType**; for details, see *eliminateSystemDependentCode*.

### Related reference

"EGL library VGLib" on page 1047

"eliminateSystemDependentCode" on page 476

"systemType" on page 1074

## VGTDLI()

The system function **VGTDLI()** uses the CBLTDLI interface to invoke a DL/I function directly. It differs from the DLILib function EGLTDLI() only in that it uses an index variable to reference the PCB for the call, whereas EGLTDLI() uses the name of the PCB record. EGL includes this function to ensure compatibility with the CSPTDLI() function in VisualAge Generator.

```
VGLib.VGTDLI() (  
    func CHAR(4) in,  
    pcbindex SMALLINT in  
    parms... ANY in)
```

*func*

A four-character DL/I function name such as ISRT or GHNP

*pcbindex*

A zero-based index following the lexical order of the fields in the program PSB (ignoring redefining fields), referencing a PCB

*parms...*

A complete list of parameters, matching in number and type those that the given DL/I function requires

To invoke a DL/I function using the AIBTDLI interface, use **DLILib.AIBTDLI()**.

### Related reference

"EGL library DLILib" on page 929

"AIBTDLI()" on page 930

"EGLTDLI()" on page 930

---

## System variables outside of EGL libraries

A variable contained in an EGL library is global to the run unit. Other system variables have different scoping characteristics and are categorized as follows:

### ConverseVar

Variables that are useful primarily in textUI applications.

### DliVar

Variables that contain information about the most recent DL/I database I/O.

Those variables are made available if you specify the program property **@DLI**.

### SysVar

Variables that are useful for general purposes.

### VGVar

Variables that are useful primarily in applications migrated from VisualAge Generator.

If you are referring to the system variable when you have another, same-named identifier in scope, you must include the category name as a qualifier. For example, you must specify **ConverseVar.eventKey** rather than **eventKey** if a second variable named **eventKey** is in scope. If a same-named identifier is not in scope, the qualifier is optional.

### Related concepts

"References to variables in EGL" on page 59

"Scoping rules and "this" in EGL" on page 57

### Related reference

"@DLI" on page 322

"ConverseVar"

"DLIVar" on page 1062

"SysVar" on page 1063

"VGVar" on page 1077

## ConverseVar

The qualifier **ConverseVar** can precede the name of each EGL system variable listed in the next table. These variables are useful primarily in textUI applications.

System variable	Description
commitOnConverse	Specify whether a commit and a release of resources occurs in a text application, before a non-segmented program issues a converse. The default value is 0 (meaning <i>no</i> ) for non-segmented programs and 1 (meaning <i>yes</i> ) for segmented programs.
eventKey	Identifies the key that the user pressed to return a text form to an EGL program
printerAssociation	Allows you to specify, at run time, the output destination when you print a print form.
segmentedMode	Used in a text application to change the effect of the converse statement, but the variable is ignored for this purpose in called programs.
validationMsgNum	Contains the value assigned by <b>ConverseLib.validationFailed</b> in a text application, so you can determine if a validation function reported an error.

### Related concepts

"References to variables in EGL" on page 59

"Scoping rules and "this" in EGL" on page 57

### Related reference

"System variables outside of EGL libraries" on page 1056

## commitOnConverse

The system variable **ConverseVar.commitOnConverse** specifies whether a commit and a release of resources occurs in a text application, before a non-segmented program issues a converse. The default value is 0 (meaning *no*) for non-segmented programs and 1 (meaning *yes*) for segmented programs.

You can use **ConverseVar.commitOnConverse** in any of these ways:

- As the source or target of an assignment or **move** statement
- As the variable in a logical expression used in a **case**, **if**, or **while** statement
- As the argument in a **return** or **exit** statement

Other characteristics of **ConverseVar.commitOnConverse** are as follows:

**Primitive type**

NUM

**Data length**

1

**Is value always restored after a converse?**

Yes

For details on using this variable, see *Segmentation*.

**Related concepts**

"Segmentation in text applications" on page 189

**Related reference**

"converse" on page 672

"System variables outside of EGL libraries" on page 1056

**eventKey**

The system variable **ConverseVar.eventKey** identifies the key that the user pressed to return a text form to an EGL program. The value is reset each time that the program runs the **converse** statement.

If the EGL code has no input form, the initial value of **ConverseVar.eventKey** is **ENTER**.

The following values are valid (whether uppercase, lowercase, or a combination):

- **ENTER**
- **BYPASS** (which refers to any of the keys that were specified as bypass keys for the form; or if none were specified for the form, any of the keys that were specified as bypass keys for the formGroup; or if none were specified for the formGroup, any of the keys that were specified as bypass keys for the program)
- **PA1** through **PA3**
- **PF1** through **PF24** (as also used for F1 through F24)
- **PAKEY** (for any PA key)
- **PFKEY** (for any PF or F key)

**Note:** PA keys are always treated as bypass keys.

You can use **ConverseVar.eventKey** as an operand in an **if** or **while** statement.

The characteristics of this system variable are as follows:

**Primitive type**

CHAR

**Data length**

1

**Value saved across segments**

No

**ConverseVar.eventKey** is not valid in a basic program.

**Example:** The comparison operator for **ConverseVar.eventKey** is either *is* or *not*, as in this example:

```

if (ConverseVar.eventKey IS PF3)
  exit program(0);
end

```

### Related reference

“Logical expressions” on page 593

“System variables outside of EGL libraries” on page 1056

## printerAssociation

The system variable **ConverseVar.printerAssociation** allows you to specify, at run time, the output destination when you print a print form.

You can use this variable in any of these ways:

- As the source or target in an assignment or move statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **ConverseVar.printerAssociation** are as follows:

### Primitive type

CHAR

### Data length

Varies by file type

### Is value always restored after a converse?

Yes

**ConverseVar.printerAssociation** is initialized to the system resource name specified during generation or for debugging. If a program passes control to another program, the value of **ConverseVar.printerAssociation** is set to the default value for the receiving program.

Even though multiple print jobs are allowed for a given print form, the close statement closes only the file related to the current value of **ConverseVar.printerAssociation**.

**Details specific to Java output:** For Java output, you set **ConverseVar.printerAssociation** to a two-part string with an intervening colon:

*jobID:destination*

*jobID* A sequence of characters (without a colon) that uniquely identifies each print job. The characters are case sensitive (*job01* is different from *JOB01*), and you can reuse *jobID* after a print job closes.

You can use different jobs to promote a different kind of output or a different ordering of output, depending on the flow of events in your code. Consider the following sequence of EGL statements, for example:

```

ConverseVar.printerAssociation = "job1";
print form1;
ConverseVar.printerAssociation = "job2";
print form2;
ConverseVar.printerAssociation = "job1";
print form3;

```

When the program ends, two print jobs are created:

- form1 followed by form3
- form2 alone

### *destination*

The printer or file that receives the output.

The string *destination* is optional and is ignored if the print job is still open. The following statements apply if the string is absent:

- You can omit the colon that precedes *destination*
- In most cases, the program shows a print preview dialog from which the user can specify a printer or a file for output. The exception occurs if the curses library is used on UNIX; in that case, the print job goes to the default printer.

The following statements apply to the setting of *destination* when you are generating for Windows 2000/NT/XP:

- To send output to the default printer, do as follows--
  - Specify a value that matches the **fileName** property in the resource associations part.
  - Change the Java runtime properties so that *spool* (rather than *seqws*) is the value of the related file type. For example, in the resource associations part, if the value of the **fileName** property is *myFile* and the value of **systemName** is *printer*, you must change the settings of Java runtime properties so that `vgj.ra.myFile.fileType` is set to *spool* rather than *seqws*. After your change, the properties are as follows:

```
vgj.ra.myFile.systemName=printer
vgj.ra.myFile.fileType=spool
```
- To send output to a file, specify a value that matches the **fileName** property in the resource associations part, when *seqws* is the value of the related **fileType** property in the resource associations part. The **systemName** property in the resource associations part contains the name of the operating-system file that receives the output.
- Do not specify the value *printer* as the value of *destination*. If you do, the print preview dialog is displayed to the user, but that behavior may change in later versions of EGL.

The following statements apply to the setting of *destination* when you are generating for UNIX:

- To send output to the default printer (regardless of whether the curses library is in use), specify a value that matches the **fileName** property in the resource associations part, when *spool* is the value of the related **fileType** property in the resource associations part.
- To send output to a file, specify a value that matches the **fileName** property in the resource associations part, when *seqws* is the value of the related **fileType** property in the resource associations part. The **systemName** property in the resource associations part contains the name of the operating-system file that receives the output.
- Do not specify the value *printer* as the value of *destination*. If you do (and if the curses library is not in use), the print preview dialog is displayed to the user, but that behavior may change in later versions of EGL.

## segmentedMode

The system variable **ConverseVar.segmentedMode** is used in a text application to change the effect of the converse statement, but the variable is ignored for this purpose in called programs. For background information, see *Segmentation in text applications*.

Values of **ConverseVar.segmentedMode** are as follows:

- 1 The next **converse** statement runs in segmented mode.
- 0 The next **converse** statement runs in non-segmented mode.

The default value is 0 for non-segmented programs and 1 for segmented programs. The variable is reset to the default after the **converse** statement runs.

You can use this variable in any of these ways:

- As the source or destination in an assignment or move statement
- As the count value in a **move...for count** statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **ConverseVar.segmentedMode** are as follows:

### Primitive type

NUM

### Data length

1

### Is value restored after a converse?

No

### Related concepts

"Segmentation in text applications" on page 189

### Related reference

"System variables outside of EGL libraries" on page 1056

## validationMsgNum

The system variable **ConverseVar.validationMsgNum** contains the value assigned by **ConverseLib.validationFailed** in a text application, so you can determine if a validation function reported an error. The value is reset to zero in each of the following cases:

- The program initializes
- The program issues a converse, display, or print statement
- The program reissues a converse statement to display a text form as the result of a validation error

You can use **ConverseVar.validationMsgNum** in these ways:

- As the source or target of an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As the variable in a logical expression
- As the argument in a **return** or **exit** statement



The characteristics of **ConverseVar.validationMsgNum** are as follows:

**Primitive type**

INT

**Is value always restored after a converse?**

No

**Example**

```
/*Keep the first message number that was set
   during validation routines */
if (ConverseVar.validationMsgNum > 0)
    ConverseLib.validationFailed(10);
end
```

**Related reference**

"converse" on page 672

"validationFailed()" on page 918

"display" on page 676

"print" on page 737

"System variables outside of EGL libraries" on page 1056

## DLIVar

The **DLIVar** variables contain information about the most recent DL/I database I/O. These variables are available only if you assign the complex property **@DLI** to your program.

Each DL/I I/O call changes the values in all the **DLIVar** variables other than **DLIVar.handleHardDLIErrors**, which retains its value until you explicitly reset it.

The variables are as follows::

**dbName char(8)**

The name of the DL/I database accessed on the last call.

**segmentLevel num(2)**

The level number of the lowest segment accessed on the most recent DL/I call (root level is 01). If DL/I was unable to retrieve the requested segment, this is the level from the last successful call.

**statusCode char(2)**

The DL/I status code for the last I/O call, such as GA or II.

**procOptions char(4)**

Contains the DL/I options for the database accessed by the last DL/I I/O call.

**segmentName char(8)**

The name of the lowest segment accessed on the most recent DL/I I/O call.

**keyAreaLen int**

The number of bytes of the **keyArea** field actually used on the most recent I/O call.

**keyArea char(32767)**

This is the unformatted data returned from the database segment. The usable data is located in the portion of the key area specified by **keyArea[1 : DLIVar.keyAreaLen]**.

**numSensitiveSegs int**

The number of segment types to which a program is sensitive for the database accessed during the last DL/I I/O function.

**cicsError char(2)**

CICS error code (if any) on the most recent DL/I I/O call. For more information about return codes, refer to the CICS application reference for your version of CICS.

**cicsCondition char(2)**

CICS condition code (if any) on the most recent DL/I I/O call, as follows:

00

Normal response

08 Request was not valid

0C Not open

**cicsRestart num(1)**

On CICS, if this field is equal to 1, the DL/I program was restarted during the last DL/I I/O function. If 0, the program was not restarted.

**handleHardDLIErrors**

Controls whether a program continues to run after a hard error occurs on a DL/I or IMS I/O operation in a **try** block; however, if **VGVar.handleHardIOErrors** is set to 1, **DLIVar.handleHardDLIErrors** has no effect because the program continues to run after any hard error,

The default value of **DLIVar.handleHardDLIErrors** is 1, unless you set the program property **handleHardDLIErrors** to *no*, which sets the variable to 0. (The property **handleHardDLIErrors** is available for programs and other generatable logic parts.)

You can use **VGVar.handleHardDLIErrors** in any of these ways:

- As the source or target of an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As the variable in a logical expression used in a **case**, **if**, or **while** statement
- As the argument in a **return** or **exit** statement

The characteristics of **DLIVar.handleHardDLIErrors** are as follows:

**Primitive type**

NUM

**Data length**

1

**Is value always restored after a converse?**

Yes

For other details, see *Exception handling*.

**Related concepts**

"DL/I database support" on page 310

**Related reference**

"@DLI" on page 322

"Exception handling" on page 94

"PCB record part properties" on page 145

## SysVar

The qualifier **SysVar** can precede the name of each EGL system variable listed in the next table. These variables are useful for various purposes.

System variable	Description
arrayIndex	<p>Contains a number:</p> <ul style="list-style-type: none"> <li>• The number of the first element in an array that matches the search condition of a simple logical expression with an <b>in</b> operator.</li> <li>• Zero, if no array element matches the search condition.</li> <li>• The number of the last element modified in the target array after a <b>move ... for count</b> statement.</li> </ul>
callConversionTable	<p>Contains the name of the conversion table that is used to convert data when your program does the following at run time:</p> <ul style="list-style-type: none"> <li>• Passes arguments in a call to a program on a remote system</li> <li>• Passes arguments when invoking a remote program by way of the system function <code>sysLib.startTransaction</code></li> <li>• Accesses a file at a remote location</li> </ul>
errorCode	<p>Receives a status code after any of the following events:</p> <ul style="list-style-type: none"> <li>• The invocation of a call statement, if that statement is in a try block</li> <li>• An I/O operation on an indexed, MQ, relative, or serial file</li> <li>• The invocation of almost any system function in these cases-- <ul style="list-style-type: none"> <li>– The invocation is within a try block; or</li> <li>– The program is running in VisualAge Generator Compatibility mode and <b>VGVar.handleSysLibraryErrors</b> is set to 1</li> </ul> </li> </ul>
formConversionTable	<p>Contains the name of the conversion table that is used for bidirectional text conversion when an EGL-generated Java program acts as follows:</p> <ul style="list-style-type: none"> <li>• Shows a text or print form that includes a series of Hebrew or Arabic characters; or</li> <li>• Shows a text form that accepts a series of Hebrew or Arabic characters from a user.</li> </ul>
overflowIndicator	<p>Is set to 1 when arithmetic overflow occurs. By checking the value of this variable, you can test for overflow conditions.</p>
returnCode	<p>Contains an external return code, as set by your program and made available to the operating system.</p>
sessionID	<p>Contains an ID that is specific to the Web application server session.</p>
sqlca	<p>Contains the entire SQL communication area (SQLCA).</p>

System variable	Description
sqlcode	Contains the return code for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.
sqlState	Contains the SQL state value for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.
systemType	Identifies the target system in which the program is running.
terminalID	Is initialized from the Java Virtual Machine system property <i>user.name</i> and is blank if the property cannot be retrieved.
transactionID	As described in the topic <i>transactionID</i> .
transferName	Allows you to specify, at run time, the name of the program or transaction to which you want to transfer.
userID	Contains a user identifier in environments where one is available.

### Related concepts

"References to variables in EGL" on page 59

"Scoping rules and "this" in EGL" on page 57

### Related reference

"System variables outside of EGL libraries" on page 1056

## arrayIndex

The system variable **SysVar.arrayIndex** contains a number:

- The number of the first element in an array that matches the search condition of a simple logical expression with an **in** operator, as shown in a later example.
- Zero, if no array element matches the search condition.
- The number of the last element modified in the target array after a **move ... for count** statement.

You can use **SysVar.arrayIndex** as any of these:

- As an array subscript to access the matching row or array element
- As the source or target in an assignment or **move** statement
- As the count value in a **move ... for count** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.arrayIndex** are as follows:

### Primitive type

BIN

### Data length

4

### Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

**Example:** Assume that the record *myRecord* is based on the following part:

```
Record mySerialRecPart
  serialRecord:
    fileName = "myFile"
  end
  10 zipCodeArray CHAR(9)[100];
  10 cityStateArray CHAR(30)[100];
end
```

Furthermore, assume that the arrays are initialized with zip codes and city-and-state combinations.

The following code sets the variable *currentCityState* to the city and state that corresponds to the specified zip code:

```
currentZipCode = "27540";
if (currentZipCode in myRecord.zipCodeArray)
  currentCityState = myRecord.cityStateArray[SysVar.arrayIndex];
end
```

After the **if** statement, **SysVar.arrayIndex** contains the index of the first *zipCodeArray* element that contains the value of "27540". If "27540" is not found in *zipCodeArray*, the value of **SysVar.arrayIndex** is 0.

### Related concepts

"Segmentation in text applications" on page 189

### Related reference

"Arrays" on page 75

"in operator" on page 629

"Logical expressions" on page 593

"System variables outside of EGL libraries" on page 1056

## callConversionTable

The system variable **SysVar.callConversionTable** contains the name of the conversion table that is used to convert data when your program does the following at run time:

- Passes arguments in a call to a program on a remote system
- Passes arguments when invoking a remote program by way of the system function `SysLib.startTransaction`
- Accesses a file at a remote location

The conversion occurs when the data is being moved between EBCDIC-based and ASCII-based systems or between systems that use different code pages. Conversion is possible only if the linkage options part used at generation time specifies **PROGRAMCONTROLLED** as the value of property **conversionTable** in the **callLink** or **asynchLink** element. Conversion does not occur, however, if **PROGRAMCONTROLLED** is specified but **SysVar.callConversionTable** is blank.

**Characteristics:** The characteristics of **SysVar.callConversionTable** are as follows:

#### Primitive type

CHAR

#### Data length

8

## Value saved across segments?

Yes

**Definition considerations:** You should use **SysVar.callConversionTable** to switch conversion tables in a program or to turn data conversion on or off in a program.

**SysVar.callConversionTable** is initialized to blanks. To cause conversion to occur, make sure that the linkage options part includes the value **PROGRAMCONTROLLED**, as described earlier, and move the name of a conversion table to the system variable. You can set **SysVar.callConversionTable** to an asterisk (\*) to use the default conversion table for the default national language code. This setting references the default locale on the target system provided the locale is mapped to one of the languages that can be specified for the **targetNLS** build descriptor option.

Conversion is performed on the system that originates the call, invocation, or file access. When you define multiple levels of a record structure, conversion is performed on the lowest level items (the items with no substructure).

You can use **SysVar.callConversionTable** in these ways:

- As the source or target operand in an assignment or **move** statement
- As a variable in a logical expression
- As an argument in a **return** or **exit** statement

A comparison of **SysVar.callConversionTable** with another value tests true only if the match is exact. If you initialize **SysVar.callConversionTable** with a lowercase value, for example, the lowercase value matches only a lowercase value.

The value that you place in **SysVar.callConversionTable** remains unchanged for purposes of comparison.

### Example:

```
SysVar.callConversionTable = "ELACNENU";  
// conversion table for US English COBOL generation
```

### Related reference

"Data conversion" on page 558

"startTransaction()" on page 1041

"System variables outside of EGL libraries" on page 1056

"targetNLS" on page 495

## conversationID

The system variable **SysVar.conversationID** stores the conversation ID, which is assigned when an EGL program uses a VGUI record to present a Web page. A related variable is *sessionID*, which contains an ID that is specific to the Web application server session.

The conversation ID is unchanged when a program in a Web application is invoked by way of a **converse** statement or by way of a **show** statement that has a returning clause. A new conversation ID is assigned, however, when the user invokes a program in response to a **show** statement that has no returning clause.

You can use **SysVar.conversationID** in these ways:

- As the source in an assignment or **move** statement

- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.conversationID** are as follows:

**Primitive type**

CHAR

**Data length**

8 (padded with blanks if the value has less than 8 characters)

**Is value restored after a converse?**

Yes

**Example:**

```
item10 = SysVar.conversationID;
```

**Related concepts**

**Related reference**

“sessionID” on page 1071

“SysVar” on page 1063

**errorCode**

The system variable **SysVar.errorCode** receives a status code after any of the following events:

- The invocation of a call statement, if that statement is in a try block
- An I/O operation on an indexed, MQ, relative, or serial file
- The invocation of almost any system function in these cases--
  - The invocation is within a try block; or
  - The program is running in VisualAge Generator Compatibility mode and **VGVar.handleSysLibraryErrors** is set to 1

The **SysVar.errorCode** values associated with a given system function are described in relation to the system function, not in the current topic.

You can use **SysVar.errorCode** in these ways:

- As the source or target in an assignment or **move** statement
- As a variable in a logical expression
- In a function invocation, as an argument associated with an in, out, or inOut parameter

**SysVar.errorCode** is set to 0 if the call, I/O, or system function invocation is successful.

The characteristics of **SysVar.errorCode** are as follows:

**Primitive type**

CHAR

**Data length**

8

**Is value always restored after a converse?**

Yes

For an overview that includes details on **SysVar.errorCode**, see *Exception handling*. The list of possible **SysVar.errorCode** values is provided in *EGL Java runtime error codes*.

**Example:**

```
if (SysVar.errorCode == "00000008")
  exit program;
end
```

**Related reference**

"EGL Java runtime error codes" on page 1097

"Exception handling" on page 94

"System variables outside of EGL libraries" on page 1056

"try" on page 754

"handleSysLibraryErrors" on page 1084

**formConversionTable**

The system variable **SysVar.formConversionTable** contains the name of the conversion table that is used for bidirectional text conversion when an EGL-generated Java program acts as follows:

- Shows a text or print form that includes a series of Hebrew or Arabic characters; or
- Shows a text form that accepts a series of Hebrew or Arabic characters from a user.

**Characteristics:** The characteristics of **SysVar.formConversionTable** are as follows:

**Primitive type**

CHAR

**Data length**

8

**Value saved across segments?**

Yes

**Related reference**

"Bidirectional language text" on page 561

"Data conversion" on page 558

"System variables outside of EGL libraries" on page 1056

**overflowIndicator**

The system variable **SysVar.overflowIndicator** is set to 1 when arithmetic overflow occurs. By checking the value of this variable, you can test for overflow conditions.

After detection of an overflow condition, **SysVar.overflowIndicator** is not reset automatically. You must include code in your program to reset

**SysVar.overflowIndicator** to 0 before performing any calculations that may trigger overflow checks.

You can use **SysVar.overflowIndicator** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement



The characteristics of **SysVar.overflowIndicator** are as follows:

**Primitive type**

NUM

**Data length**

1

**Is value always restored after a converse?**

Yes

**Example:**

```
SysVar.overflowIndicator = 0;
VGVar.handleOverflow = 2;
a = b;
if (SysVar.overflowIndicator == 1)
    add errorrecord;
end
```

**Related reference**

"Assignments" on page 456

"System variables outside of EGL libraries" on page 1056

"handleOverflow" on page 1083

**returnCode**

The system variable **SysVar.returnCode** contains an external return code, as set by your program and made available to the operating system. It is not possible to pass return codes from one EGL program to another. A non-zero return code does not cause EGL to run an onException block, for example.

The initial value of **SysVar.returnCode** is zero, and the value must be in the range of -2147483648 to 2147483647, inclusive.

**SysVar.returnCode** is meaningful only for a main text program (which runs outside of J2EE) or a main batch program (which runs either outside of J2EE or in a J2EE application client). The purpose of **SysVar.returnCode** in this context is to provide a code for the command file or exec that invokes the program. If the program ends with an error that is not under the program's control, the EGL runtime ignores the setting of **SysVar.returnCode** and attempts to return the value 693.

You can use **SysVar.returnCode** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.returnCode** are as follows:

**Primitive type**

BIN

**Data length**

9

**Is value always restored after a converse?**

Yes

**Example:**

```
SysVar.returnValue = 6;
```

**Related reference**

“System variables outside of EGL libraries” on page 1056

**sessionID**

In a program of type VGWebTransaction, the system variable **SysVar.sessionID** contains an ID that is specific to the Web application server session. You can use the **SysVar.sessionID** value as a key value to access file or database information shared between programs. A related variable for VGWebTransaction programs is **SysVar.conversationID**.

Outside of Web applications, the following statements apply:

- The system variable **SysVar.sessionID** contains a system-dependent user identifier or terminal identifier for your program. In EGL-generated Java code, the value is from the Java Virtual machine property *user.name*.
- **SysVar.sessionID** is supported for this use only for compatibility with products that preceded EGL (specifically, for CSP releases prior to CSP 370AD Version 4 Release 1). It is recommended that you use **SysVar.userID** or **SysVar.terminalID** instead.

You can use **SysVar.sessionID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **SysVar.sessionID** are as follows:

**Primitive type**

CHAR

**Data length**

8 (padded with blanks if the value has less than 8 characters)

**Is value always restored after a converse?**

Yes

**SysVar.sessionID** is initialized from the Java Virtual Machine system property *user.name*; and if the property cannot be retrieved, **SysVar.sessionID** is blank.

**Example:**

```
myItem = SysVar.sessionID;
```

**Related reference**

“conversationID” on page 1067

“SysVar” on page 1063

“terminalID” on page 1075

“userID” on page 1076

**sqlca**

The system variable **SysVar.sqlca** contains the entire SQL communication area (SQLCA). As noted later, the current values of a subset of fields in the SQLCA are available to you after your code accesses a relational database.

You can use **SysVar.sqlca** in these ways:

- As the source or target in an assignment or **move** statement
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

In order to refer to specific fields in the SQLCA, you must move **SysVar.sqlca** to a base record. The record must have a structure as specified in the SQLCA description for your database management system. Use the base record if you pass the SQLCA contents to a remote program so that the contents will be converted correctly to the remote system data format.

For specific information about the fields that are available in **SysVar.sqlca**, refer to the following topics:

- VGVar.sqlerrd
- SysVar.sqlcode
- SysVar.sqlState
- VGVar.sqlWarn

The characteristics of **SysVar.sqlca** are as follows:

**Primitive type**

HEX

**Data length**

272 (136 bytes)

**Is value always restored after a converse?**

Only in a non-segmented text program; for details see *Segmentation*

**Example:**

```
myItem = SysVar.sqlca;
```

**Related concepts**

"Segmentation in text applications" on page 189

"SQL support" on page 277

**Related reference**

"System variables outside of EGL libraries" on page 1056

"sqlcode"

"sqlState" on page 1073

"sqlerrd" on page 1085

"sqlWarn" on page 1087

**sqlcode**

The system variable **SysVar.sqlcode** contains the return code for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

You can use **SysVar.sqlcode** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the "for count" of a **move** statement)

- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.sqlcode** are as follows:

**Primitive type**

BIN

**Data length**

9

**Is value always restored after a converse?**

Only in a non-segmented text program; for details see *Segmentation*

**Example:**

```
rcitem = SysVar.sqlcode;
```

**Related concepts**

“Segmentation in text applications” on page 189

“SQL support” on page 277

**Related reference**

“System variables outside of EGL libraries” on page 1056

**sqlState**

The system variable **SysVar.sqlState** contains the SQL state value for the most recently completed SQL I/O operation. The code is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

You can use **SysVar.sqlState** in these ways:

- As the source or target in an assignment or **move** statement
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **SysVar.sqlState** are as follows:

**Primitive type**

CHAR

**Data length**

5

**Is value always restored after a converse?**

Only in a non-segmented text program; for details see *Segmentation*

**Example:**

```
rcitem = SysVar.sqlState;
```

**Related concepts**

“Segmentation in text applications” on page 189

“SQL support” on page 277

## Related reference

“System variables outside of EGL libraries” on page 1056

## systemType

The system variable **SysVar.systemType** identifies the target system in which the program is running. If the generated output is a Java wrapper, **SysVar.systemType** is not available. Otherwise, the valid values are as follows:

**aix** For AIX  
**debug** For the EGL Debugger  
**hp** For HP-UX  
**iseriesj**  
For iSeries  
**linux** For Linux (on Intel-based hardware)  
**solaris** For Solaris  
**win** For Windows 2000/NT/XP

You can use **SysVar.systemType** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **SysVar.systemType** are as follows:

**Primitive type**  
CHAR

**Data length**  
8 (padded with blanks)

**Is value always restored after a converse?**  
Yes

Use **SysVar.systemType** instead of **VGLib.getVAGSysType**.

**Definition considerations:** The value of **SysVar.systemType** does not affect what code is validated at generation time. For example, the following **add** statement is validated even if you are generating for Windows:

```
if (sysVar.systemType IS AIX)
  add myRecord;
end
```

To avoid validating code that will never run in the target system, take either of the following actions:

- Set the build descriptor option **EliminateSystemDependentCode** to YES. In the current example, the **add** statement is not validated if you set that build descriptor option to YES. Be aware, however, that the generator can eliminate system-dependent code only if the logical expression (in this case, **SysVar.systemType IS AIX**) is simple enough to evaluate at generation time.
- Alternatively, move the statements that you do not want to validate to a second program; then, let the original program call the new program conditionally:

```
if (SysVar.systemType IS AIX)
  call myAddProgram myRecord;
end
```

**Example:**

```

if (SysVar.systemType is WIN)
  call myAddProgram myRecord;
end

```

**Related reference**

“eliminateSystemDependentCode” on page 476

“System variables outside of EGL libraries” on page 1056

“getVAGSysType()” on page 1054

**terminalID**

**SysVar.terminalID** (like **SysVar.sessionID**) is initialized from the Java Virtual Machine system property *user.name*, and if the property cannot be retrieved, **SysVar.terminalID** is blank.

You can use **SysVar.terminalID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **SysVar.terminalID** are as follows:

**Primitive type**

CHAR

**Data length**

8, padded with blanks if the value has less than the maximum number of characters

**Is value always restored after a converse?**

Yes

**Example:**

```
myItem10 = SysVar.terminalID;
```

**transactionID**

The variable is not used; but if the program was invoked by a **transfer** statement of the form *transfer to program*, the variable contains the name of the transferring program.

You can use this variable in any of these ways:

- As the source or destination in an assignment or move statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **SysVar.transactionID** are as follows:

**Primitive type**

CHAR

**Data length**

8

**Is value always restored after a converse?**

Yes

### Related concepts

“Segmentation in text applications” on page 189

### Related reference

“System variables outside of EGL libraries” on page 1056

## transferName

The system variable **SysVar.transferName** allows you to specify, at run time, the name of the program or transaction to which you want to transfer control by way of the **transfer** or **show** statement.

You can use this variable in any of these ways:

- As the source or target in an assignment or move statement
- As a program or transaction name in a transfer statement
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **SysVar.transferName** are as follows:

### Primitive type

CHAR

### Data length

8

### Is value always restored after a converse?

Yes

### Related reference

“System variables outside of EGL libraries” on page 1056

“show” on page 751

“transfer” on page 752

## userID

The system variable **SysVar.userID** contains a user identifier in environments where one is available.

You can use **SysVar.userID** in these ways:

- As the source in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in a **return** statement

The characteristics of **SysVar.userID** are as follows:

### Primitive type

CHAR

### Data length

8 (padded with blanks if the value has less than 8 characters)

### Is value always restored after a converse?

Yes

**SysVar.userID** is initialized from the Java Virtual Machine system property *user.name*; and if the property cannot be retrieved, **SysVar.userID** is blank.

**Example:**

```
myItem = SysVar.userID;
```

**VGVar**

The qualifier **VGVar** can precede the name of each EGL system variable listed in the next table. These variables are useful primarily in applications migrated from VisualAge Generator.

System variable	Description
currentFormattedGregorianDate	Contains the current system date in long Gregorian format.
currentFormattedJulianDate	Contains the current system date in long Julian format.
currentFormattedTime	Contains the current system time in HH:mm:ss format.
currentGregorianDate	Contains the current system date in eight-digit Gregorian format (yyyyMMdd).
currentJulianDate	Contains the current system date in seven-digit Julian format (yyyyDDD). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.
currentShortGregorianDate	Contains the current system date in six-digit Gregorian format (yyMMdd). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.
currentShortJulianDate	Contains the current system date in five-digit Julian format (yyDDD). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.
handleHardIOErrors	Controls whether a program continues to run after a hard error occurs on an I/O operation in a try block.
handleOverflow	Controls error processing after an arithmetic overflow.
handleSysLibraryErrors	Specifies whether the value of system variable <b>SysVar.errorCode</b> is affected by the invocation of a system function.
mqConditionCode	Contains the completion code from an MQSeries API call following an <b>add</b> or <b>get next</b> I/O operation for an MQ record.
sqlerrd	Six-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option.
sqlerrmc	Contains the substitution variables for the error message associated with the return code in <b>SysVar.sqlcode</b> .
sqlIsolationLevel	Indicates the level of independence of one database transaction from another.



System variable	Description
sqlWarn	Eleven-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater than the warning number in the SQL SQLCA description.

#### Related concepts

"References to variables in EGL" on page 59

"Scoping rules and "this" in EGL" on page 57

#### Related reference

"System variables outside of EGL libraries" on page 1056

### currentFormattedGregorianDate

The system variable **VGVar.currentFormattedGregorianDate** contains the current system date in long Gregorian format. The value is automatically updated each time system variable is referenced by your program.

The format is in this Java runtime property:

```
vgj.datemask.gregorian.long.NLS
```

#### NLS

The NLS (national language support) code specified in the Java runtime property **vgj.nls.code**. The code is one of those listed for the targetNLS build descriptor option. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java runtime properties (details)*.

The format specified in **vgj.datemask.gregorian.long.NLS** includes dd (for numeric day), MM (for numeric month), and yyyy (for numeric year), with characters other than d, M, y, or digits used as separators. You can specify the format in the **dateMask** build descriptor option, and the default format is specific to the locale.

You can use **VGVar.currentFormattedGregorianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

Make sure that this Gregorian long date format is the same as the date format specified for the SQL database manager. Matching the two formats enables **VGVar.currentFormattedGregorianDate** to produce dates in the format expected by the database manager.

The characteristics of **VGVar.currentFormattedGregorianDate** are as follows:

#### Primitive type

CHAR

#### Data length

10

#### Value saved across segments

No

#### Example:

```
myDate = VGVar.currentFormattedGregorianDate;
```

### Related concepts

“Build descriptor part” on page 383

“Java runtime properties” on page 431

### Related tasks

“Editing Java runtime properties in a build descriptor” on page 391

### Related reference

“EGL library DateTimeLib” on page 919

“Java runtime properties (details)” on page 642

“System variables outside of EGL libraries” on page 1056

“targetNLS” on page 495

## currentFormattedJulianDate

The system variable **VGVar.currentFormattedJulianDate** contains the current system date in long Julian format. The value is automatically updated each time the system variable is referenced by your program

The format is in this Java runtime property:

```
vgj.datemask.julian.long.NLS
```

### NLS

The NLS (national language support) code specified in the Java runtime property **vgj.nls.code**. The code is one of those listed for the targetNLS build descriptor option. Uppercase English (code ENP) is not supported.

For additional details on **vgj.nls.code**, see *Java runtime properties (details)*.

The format specified in **vgj.datemask.julian.long.NLS** includes DDD (for numeric day) and yyyy (for numeric year), with characters other than D, y, or digits used as separators. You can specify the format in the **dateMask** build descriptor option, and the default format is specific to the locale.

You can use **VGVar.currentFormattedJulianDate** as the source in an assignment or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentFormattedJulianDate** are as follows:

### Primitive type

CHAR

### Data length

8

### Value saved across segments

No

Uppercase English (NLS code ENP) is not supported.

### Example:

```
myDate = VGVar.currentFormattedJulianDate;
```

### Related concepts

“Build descriptor part” on page 383

“Java runtime properties” on page 431

### Related tasks

“Editing Java runtime properties in a build descriptor” on page 391

**Related reference**

"EGL library DateTimeLib" on page 919

"Java runtime properties (details)" on page 642

"System variables outside of EGL libraries" on page 1056

"targetNLS" on page 495

**currentFormattedTime**

The system variable **VGVar.currentFormattedTime** contains the current system time in HH:mm:ss format. The value is automatically updated each time it is referenced by your program.

You can use **VGVar.currentFormattedTime** in these ways:

- As the source in an assignment or **move** statement
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.currentFormattedTime** are as follows:

**Primitive type**

CHAR

**Data length**

8

**Value saved across segments**

No

**Example:**

```
timeField = VGVar.currentFormattedTime;
```

**Related reference**

"EGL library DateTimeLib" on page 919

"System variables outside of EGL libraries" on page 1056

**currentGregorianDate**

The system variable **VGVar.currentGregorianDate** contains the current system date in eight-digit Gregorian format (yyyyMMdd).

The **VGVar.currentGregorianDate** value is updated automatically before each reference. The value is numeric and contains no separator characters.

You can use **VGVar.currentGregorianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentGregorianDate** are as follows:

**Primitive type**

DATE

**Data length**

8

**Value saved across segments**

No

**Example:**

```
myDate = VGVar.currentGregorianDate
```

**Related reference**

"EGL library DateTimeLib" on page 919

"System variables outside of EGL libraries" on page 1056

**currentJulianDate**

The system variable **VGVar.currentJulianDate** contains the current system date in seven-digit Julian format (yyyyDDD). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.

The value is numeric, contains no separator characters, and is updated automatically before each reference.

You can use **VGVar.currentJulianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentJulianDate** are as follows:

**Primitive type**

NUM

**Data length**

7

**Value saved across segments**

No

**Example:**

```
myDay = VGVar.currentJulianDate;
```

**Related reference**

"EGL library DateTimeLib" on page 919

"System variables outside of EGL libraries" on page 1056

**currentShortGregorianDate**

The system variable **VGVar.currentShortGregorianDate** contains the current system date in six-digit Gregorian format (yyMMdd). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.

The **VGVar.currentShortGregorianDate** value is automatically updated each time it is referenced by the program. The returned value is numeric and contains no separator characters.

You can use **VGVar.currentShortGregorianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentShortGregorianDate** are as follows:

**Primitive type**

NUM

**Data length**

6

**Value saved across segments**

No

**Example:**

```
myDay = VGVar.currentShortGregorianDate;
```

### Related reference

“EGL library DateTimeLib” on page 919

“System variables outside of EGL libraries” on page 1056

### currentShortJulianDate

The system variable **VGVar.currentShortJulianDate** contains the current system date in five-digit Julian format (yyDDD). Avoid using this variable, which exists to support code migration from VisualAge Generator to EGL.

The value is numeric, contains no separator characters, and is automatically updated each time it is referenced by your program.

You can use **VGVar.currentShortJulianDate** as the source in an **assignment** or **move** statement or as the argument in a **return** or **exit** statement.

The characteristics of **VGVar.currentShortJulianDate** are as follows:

#### Primitive type

NUM

#### Data length

5

#### Value saved across segments

No

#### Example:

```
myDay = VGVar.currentShortJulianDate;
```

### Related reference

“EGL library DateTimeLib” on page 919

“System variables outside of EGL libraries” on page 1056

### handleHardIOErrors

The system variable **VGVar.handleHardIOErrors** controls whether a program continues to run after a hard error occurs on an I/O operation in a try block. The default value is 1, unless you set the program property **handleHardIOErrors** to *no*, which sets the variable to 0. (That property is also available for other generatable logic parts.) For background information, see *Exception handling*.

You can use **VGVar.handleHardIOErrors** in any of these ways:

- As the source or target of an assignment or **move** statement (also allowed in the “for count” of a **move** statement)
- As the variable in a logical expression used in a **case**, **if**, or **while** statement
- As the argument in a **return** or **exit** statement

The characteristics of **VGVar.handleHardIOErrors** are as follows:

#### Primitive type

NUM

#### Data length

1

#### Is value always restored after a converse?

Yes

## Example

```
VGVar.handleHardIOErrors = 1;
```

### Related reference

“Exception handling” on page 94

“System variables outside of EGL libraries” on page 1056

## handleOverflow

The system variable **VGVar.handleOverflow** controls error processing after an arithmetic overflow. Two types of overflow conditions are detected:

- *User variable overflow* occurs when the result of an arithmetic operation or assignment to a numeric item causes a significant value (not decimal positions) to be lost due to the length of the item.
- *Maximum value overflow* occurs when the result of an arithmetic operation is greater than 18 digits.

You can set **VGVar.handleOverflow** to one of the following values. (The default setting is 0.)

Value	Effect on user overflow	Effect on maximum value overflow
0	The program sets the system variable <b>SysVar.overflowIndicator</b> to 1 and continues	The program ends with an error message
1	The program ends with an error message	The program ends with an error message
2	The program sets the system variable <b>SysVar.overflowIndicator</b> to 1 and continues	The program sets the system variable <b>SysVar.overflowIndicator</b> to 1 and continues

You can use **VGVar.handleOverflow** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the “for count” of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.handleOverflow** are as follows:

### Primitive type

NUM

### Data length

1

### Is value always restored after a converse?

Yes

### Example:

```
VGVar.handleOverflow = 2;
```

### Related reference

“Assignments” on page 456

“System variables outside of EGL libraries” on page 1056  
“overflowIndicator” on page 1069

## handleSysLibraryErrors

The system variable **VGVar.handleSysLibraryErrors** specifies whether the value of system variable **SysVar.errorCode** is affected by the invocation of a system function. However, **VGVar.handleSysLibraryErrors** is available only when VisualAge Generator compatibility is in effect, as explained in *Compatibility with VisualAge Generator*.

For details and restrictions, see *Exception handling*.

You can use **VGVar.handleSysLibraryErrors** in these ways:

- As the source or target in an assignment or **move** statement
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.handleSysLibraryErrors** are as follows:

**Primitive type**  
NUM

**Data length**  
1

**Is value always restored after a converse?**

Only in a non-segmented text program; for details see *Segmentation*

**Example:**

```
VGVar.handleSysLibraryErrors = 1;
```

**Related concepts**

“Compatibility with VisualAge Generator” on page 532

“Segmentation in text applications” on page 189

**Related reference**

“Exception handling” on page 94

“System variables outside of EGL libraries” on page 1056

“errorCode” on page 1068

## mqConditionCode

The system variable **VGVar.mqConditionCode** contains the completion code from an MQSeries API call following an **add** or **get next** I/O operation for an MQ record. Valid values and their related meanings are as follows:

00 OK

01 WARNING

02 FAILED

You can use **VGVar.mqConditionCode** in these ways:

- As the source or target in an assignment or **move** statement (also allowed in the “for count” of a **move** statement)
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.mqConditionCode** are as follows:

**Primitive type**

NUM

**Data length**

2

**Is value always restored after a converse?**

Yes

**Example:**

```
add MQRecord;
if (VGVar.mqConditionCode == 0)
  // continue
else
  exit program;
end
```

**Related concepts**

“MQSeries support” on page 336

**Related reference**

“Exception handling” on page 94

“System variables outside of EGL libraries” on page 1056

**sqlerrd**

The system array **VGVar.sqlerrd** is a 6-element array, where each element contains the corresponding SQL communication area (SQLCA) value that was returned from the last SQL I/O option. The value in **VGVar.sqlerrd[3]**, for example, is the third value and indicates the number of rows processed for some SQL requests.

Of the elements in **VGVar.sqlerrd**, only **VGVar.sqlerrd[3]** is refreshed by the database management system for Java code or at debugging time.

You can use a **VGVar.sqlerrd** element in these ways:

- As the source or target in an assignment or **move** statement
- As the value in the **for count** clause of a **move** statement
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a variable in a logical expression
- As the argument in an **exit** or **return** statement

The characteristics of each element in the **VGVar.sqlerrd** array are as follows:

**Primitive type**

BIN

**Data length**

9

**Is value always restored after a converse?**

Only in a non-segmented text program; for details see *Segmentation*

**Example:**

```
myItem = VGVar.sqlerrd[3];
```



**Related concepts**

“Segmentation in text applications” on page 189

“SQL support” on page 277

**Related reference**

“System variables outside of EGL libraries” on page 1056

**sqlerrmc**

The system variable **VGVar.sqlerrmc** contains the error message associated with the return code in **SysVar.sqlcode**. **VGVar.sqlerrmc** is obtained from the SQL communications area (SQLCA) and can vary with the relational database manager.

**VGVar.sqlerrmc** has no meaning for the JDBC environment.

You can use **VGVar.sqlerrmc** in these ways:

- As the source or target in an assignment or **move** statement
- As a variable in a logical expression
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As the argument in an **exit** or **return** statement

The characteristics of **VGVar.sqlerrmc** are as follows

**Primitive type**

CHAR

**Data length**

70

**Is value always restored after a converse?**

Only in a non-segmented text program; for details see *Segmentation*

**Example:**

```
myItem = VGVar.sqlerrmc;
```

**Related concepts**

“Segmentation in text applications” on page 189

“SQL support” on page 277

**Related reference**

“sqlca” on page 1071

“System variables outside of EGL libraries” on page 1056

**sqlIsolationLevel**

The system variable **VGVar.sqlIsolationLevel** indicates the level of independence of one database transaction from another.

For an overview of isolation level and of the phrases *repeatable read* and *serializable transaction*, see the JDBC documentation available from Sun Microsystems, Inc.

**VGVar.sqlIsolationLevel** is for use only in programs migrated from VisualAge Generator and EGL 5.0. The function is supported (at development time) if the EGL preference **VisualAge Generator Compatibility** is selected or (at generation time) if the build descriptor option **VAGCompatibility** is set to *yes*.

For new development, set the SQL isolation level in the **SysLib.connect**.

The following values of **VGVar.sqlIsolationLevel** are in order of increasing strictness:

**0 (the default)**

Repeatable read

**1** Serializable transaction

You can use this variable in any of these ways:

- As the source or target in an assignment or move statement
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As a comparison value in a logical expression
- As the value in a return statement

Characteristics of **SysVar.transactionID** are as follows:

**Primitive type**

NUM

**Data length**

1

**Is value always restored after a converse?**

Yes

**Related reference**

"connect()" on page 1025

"System variables outside of EGL libraries" on page 1056

## **sqlWarn**

The system array **VGVar.sqlWarn** is an 11-element array, where each element contains a warning byte returned in the SQL communications area (SQLCA) for the last SQL I/O operation and where the index is one greater than the warning number in the SQL SQLCA description. The system variable **VGVar.sqlWarn[2]**, for example, refers to SQLWARN1, which indicates whether characters in an item were truncated in the I/O operation.

Of the elements in **VGVar.sqlWarn**, only the system variable **VGVar.sqlWarn[2]** is refreshed by the database management system for Java code or at debugging time.

You can use **VGVar.sqlWarn** in these ways:

- As the source or target in an assignment or **move** statement
- As the value in the **for count** clause of a **move** statement
- As a variable in a logical expression
- In a function invocation, as an argument associated with an in, out, or inOut parameter
- As the argument in an **exit** or **return** statement

The characteristics of each element in the **VGVar.sqlWarn** array are as follows:

**Primitive type**

CHAR

## Data length

1

### Is value always restored after a converse?

Only in a non-segmented text program; for details see *Segmentation*

**Definition considerations:** `VGVar.sqlWarn[2]` contains *W* if the last SQL I/O operation caused the database manager to truncate character data items because of insufficient space in the program's host variables. You can use logical expressions to test whether the values in specific host variables were truncated. For details, see the references to **trunc** in *Logical expressions*.

When the host variable is a number, no truncation warning is given. Fractional parts of a number are truncated with no indication.

**Example:** In the following example, *my-char-field* is a field in the SQL row record just processed and *lost-data* is a function that sets an error message indicating that information for *my-char-field* was truncated.

```
if (VGVar.sqlWarn[2] == 'W')
  if (my-char-field is trunc)
    lost-data();
  end
end
```

### Related concepts

"Segmentation in text applications" on page 189

"SQL support" on page 277

### Related reference

"Logical expressions" on page 593

"System variables outside of EGL libraries" on page 1056

---

## transferToTransaction element

A *transferToTransaction* element of a linkage options part specifies how a generated program transfers control to a transaction and ends processing. The element includes the property `toPgm` and may include these properties:

- `alias`, as is necessary if your code is transferring to a program whose runtime name is different from the name of the related program part
- `externallyDefined`, as is necessary if your code is transferring to a program that was not generated with EGL or VisualAge Generator

You can avoid specifying a **transferToTransaction** element when the target program is generated with VisualAge Generator or (in the absence of an `alias`) with EGL.

### Related concepts

"Linkage options part" on page 399

### Related tasks

"Adding a linkage options part to an EGL build file" on page 401

"Editing the transfer-related elements of a linkage options part" on page 404

### Related reference

"alias in transfer-related linkage elements" on page 1089

"externallyDefined in transferToTransaction element" on page 1089

## alias in transfer-related linkage elements

In the transfer-related elements of the linkage options part, the property **alias** specifies the runtime name of the program that is identified in property **toPgm**.

The value of this property must match the alias (if any) you specified when declaring the program to which you are transferring. If you did not specify an alias when declaring that program, either set the property **alias** to the name of the program part or do not set the property at all.

### Related concepts

"Linkage options part" on page 399

### Related tasks

"Adding a linkage options part to an EGL build file" on page 401

"Editing the transfer-related elements of a linkage options part" on page 404

### Related reference

"transferToTransaction element" on page 1088

## externallyDefined in transferToTransaction element

The linkage options part, transferToTransaction element, property **externallyDefined** indicates whether you are transferring to a program that was produced by software other than EGL or VisualAge Generator. Valid values are **no** (the default) and **yes**.

If you specify **yes**, an XCTL implements the transfer statement in all COBOL target systems.

If the program property **VAGCompatibility** is set to *yes*, you can specify **externallyDefined** in the transfer statement, as noted in *Compatibility with VisualAge Generator*. It is recommended that the value be specified in the transferToTransaction element instead, but the value is in effect if specified in either place.

### Related concepts

"Compatibility with VisualAge Generator" on page 532

### Related tasks

"Adding a linkage options part to an EGL build file" on page 401

"Editing the transfer-related elements of a linkage options part" on page 404

### Related reference

"transfer" on page 752

---

## VGUIRecord part in EGL source format

An example of a VGUIRecord part is as follows:

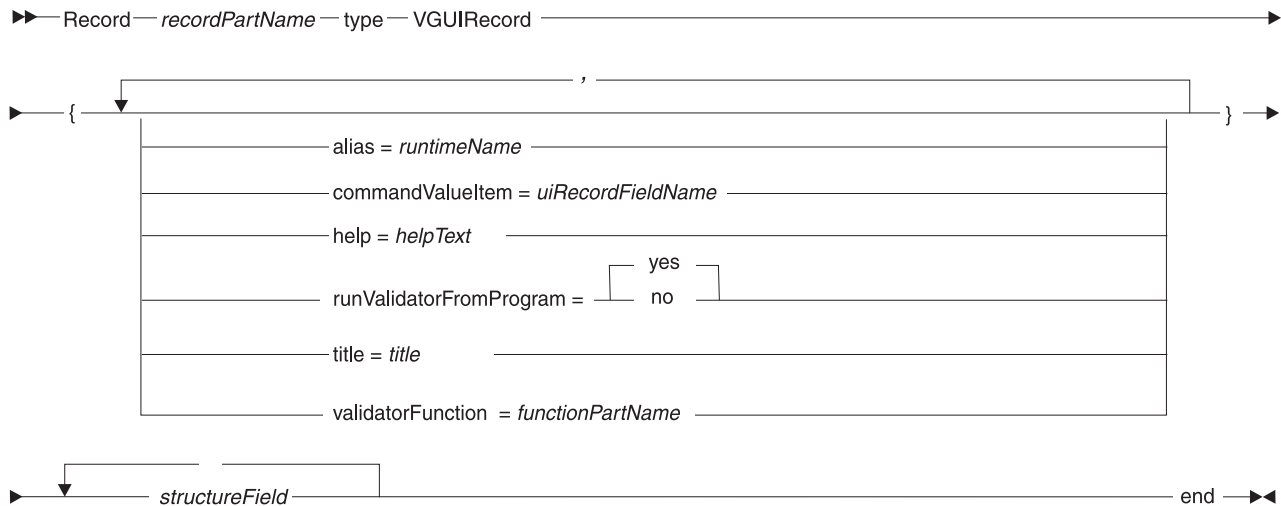
```
Record MyUIRecord Type VGUIRecord
{ commandValueItem="buttonValue" }
  10 formItem char(20)
  { uiType=uiForm,
    @programLinkData { programName="NewPgm",
                      newWindow=yes,
```

```

        uiRecordName="NewUIR",
linkParms = [
    @LinkParameter { name="key1",
        value="value1" },
    @LinkParameter { name = "key2",
        valueRef="refItem" }
]
    }
};

```

The syntax diagram for a VGUIRecord part is as follows:



#### **Record** *recordPartName* **VGUIRecord ... end**

Identifies the part as being of type VGUIRecord and specifies the part name. For the rules of naming, see *Naming conventions*.

#### **alias** = "*alias*"

A string that is incorporated into the names of generated output. If you do not set the **alias** property, the part name is used instead.

#### **commandValueItem** = "*VGUIRecordFieldName*"

Name of the VGUI record field that contains the value of the SUBMIT button or hypertext link clicked by the user. The field must be of a character type, as described in *Primitive types*.

Regardless of whether you specify a value for the property **commandValueItem**, the following is true:

- If the user clicks a SUBMIT button that has a valid `ConverseVar.eventKey` value ("PF1" - "PF24", "PA1" - "PA3", or "ENTER"), the value is placed in `ConverseVar.eventKey`
- If the user clicks a SUBMIT button that has a value other than a valid `ConverseVar.eventKey` value, the value "ENTER" is placed in `ConverseVar.eventKey`
- You can test the value of `ConverseVar.eventKey` in your code

#### **help** = "*helpText*"

A string that is available at runtime, but you must tailor the VGUI record JSP file to access that text from the VGUI record bean. You may want to include a client-side script in the VGUI record JSP so the user can easily access the help text.

**runValidatorFromProgram = yes, runValidatorFromProgram=no**

Concerns the validator function, which is the function referenced in the **validatorFunction** property.

If the property **runValidatorFromProgram** is set to *yes* (the default), the validator function runs in the EGL program that gets control after a **converse** or **show** statement is processed. Otherwise,, the validator function runs on the Web application server.

Accept the default setting if validation requires access to program variables or to other resources that are not available in the Web application server.

**title = " defaultTitle"**

Refers to the title that is associated with the VGUI record on the Web page.

You specify the title by specifying a quoted string in place of *title*; then, by default, the title is shown in an HTML <H1> tag that precedes the VGUI record. You can override and reformat the title when you customize the JSP file in PageDesigner.

**validatorFunction = "functionPartName"**

The name of the validator function, which validates input data after field-specific validation is complete.

*structureField*

A structure field, as described in *Structure field in EGL source format*. When defining that structure, consider (especially) the following primitive field-level properties:

- @programLinkData
- alias
- displayName
- help
- numElementsItem
- selectedIndexItem
- uiType

For details, see *Primitive field-level properties*.

**Related reference**

"Naming conventions" on page 778

"Primitive field-level properties" on page 793

"Primitive types" on page 34

"Structure field in EGL source format" on page 880

"setError()" on page 1037

"eventKey" on page 1058

---

## Use declaration

This section describes the use declaration, followed by details on how to write the declaration:

- "In a program or library part" on page 1092
- "In a formGroup part" on page 1094
- "In a pageHandler part" on page 1095

## Background

The use declaration allows you to easily reference data areas and functions in parts that are separately generated. A program, for instance, can issue a use declaration

that allows for easy reference to a data table, library, or form group, but only if those parts are visible to the program part. For details on visibility, see *References to parts*.

In most cases, you can reference data areas and functions from another part regardless of whether a use declaration is in effect. For example, if you are writing a program and do not have a use declaration for a library part called `myLib`, you can access the library variable called `myVar` as follows:

```
myLib.myVar
```

If you include the library name in a use declaration, however, you can reference the variable as follows:

```
myVar
```

The previous, short form of the reference is valid only if the symbol `myVar` is unique for every variable and structure item that is global to the program. (If the symbol is not unique, an error occurs.) Also, the symbol `myVar` refers to an item in the library only if a local variable or parameter does not have the same name. (A local data area takes precedence over a same-named, program-global data area.)

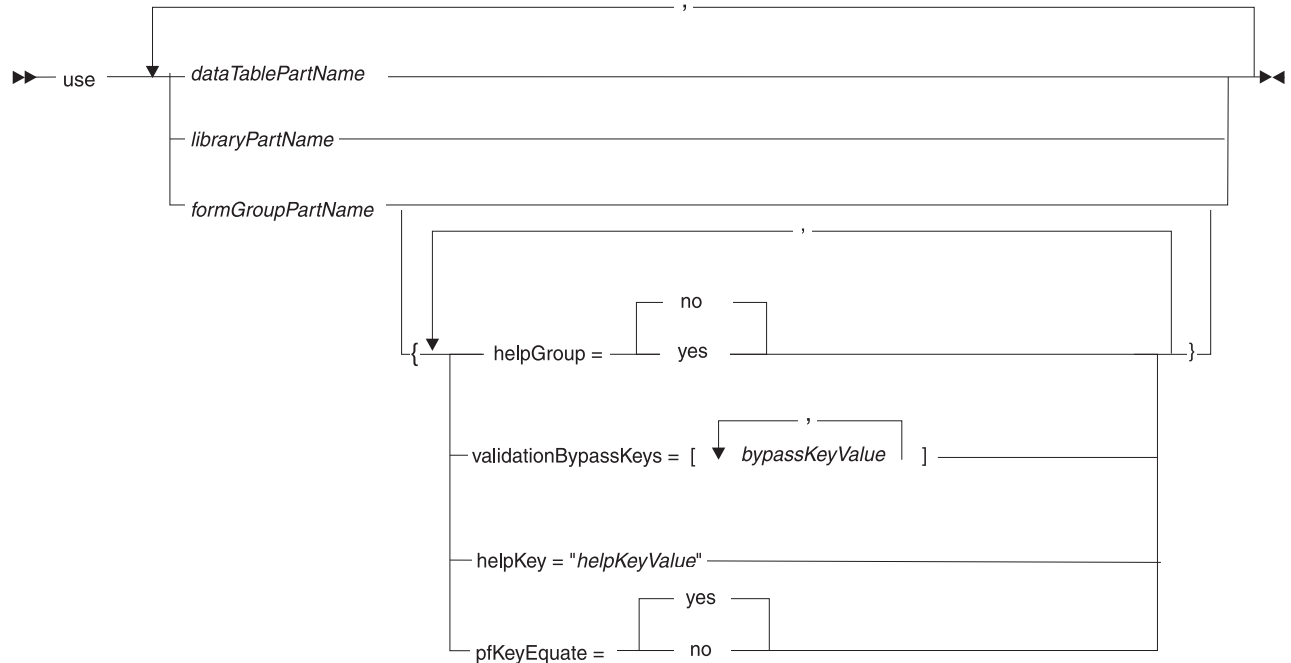
A use declaration is required in these situations:

- A program or library that uses any of the forms in a given `formGroup` part must have a use declaration for that `formGroup` part
- A `formGroup` part must have a use declaration for a form that is required by the program or library but is not embedded in the `formGroup` part
- If you have declared a function at the top level of an EGL source file rather than physically inside a container (a program, `PageHandler`, or library), that function can invoke library functions only if the following situation is in effect:
  - The container includes a use statement that refers to the library
  - In the invoking function, the property **`containerContextDependent`** is set to *yes*

Each name specified in the use declaration may be qualified by a package name, library name, or both.

## In a program or library part

Each use declaration in a program or library must be external to any function. The syntax for the declaration is as follows:



#### *dataTablePartName*

Name of a dataTable part that is visible to the program or library.

A reference in a use declaration is unnecessary for a dataTable part that is referenced in the program property **msgTablePrefix**.

You cannot override properties of a dataTable part in the use declaration.

For an overview of dataTable parts, see *DataTable part*.

#### *libraryPartName*

Name of a library part that is visible to the program or library.

You cannot override properties of the library part in the use declaration.

For an overview of library parts, see *Library part of type basicLibrary* and *Library part of type nativeLibrary*.

#### *formGroupName*

Name of a formGroup part that is visible to the program or library. For an overview of form groups, see *FormGroup part*.

A program that uses any of the forms in a given formGroup part must have a use declaration for that formGroup part.

No overrides occur for form-level properties. If a property like **validationBypassKeys** is specified in a form, for example, the value in the form is in effect at run time. If a form-level property is not specified in the form, however, the situation is as follows:



- The EGL runtime uses the value in the program's use declaration
- If no value is specified in the program's use declaration, the EGL runtime uses the value (if any) in the form group

The properties that follow let you change behaviors when a form group is accessed by a specific program.

**helpGroup = no, helpGroup = yes**

Specifies whether to use the formGroup part as a help group. The default is *no*.

**validationBypassKeys = [bypassKeyValue]**

Identifies a user keystroke that causes the EGL runtime to skip input-field validations. This property is useful for reserving a keystroke that ends the program quickly. Each *bypassKeyValue* option is as follows:

**pf*n***

The name of an F or PF key, including a number between 1 and 24, inclusive.

**Note:** Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

If you specify multiple keys, separate one from the next with a comma.

**helpKey = "helpKeyValue"**

Identifies a user keystroke that causes the EGL runtime to present a help form to the user. The *helpKeyValue* option is as follows:

**pf*n***

The name of an F or PF key, including a number between 1 and 24, inclusive.

**Note:** Function keys on a PC keyboard are often *f* keys such as f1, but EGL uses the IBM *pf* terminology so that (for example) f1 is called pf1.

**pfKeyEquate = yes, pfKeyEquate = no**

Specifies whether the keystroke that is registered when the user presses a high-numbered function key (PF13 through PF24) is the same as the keystroke that is registered when the user presses a function key that is lower by 12. The default is *yes*. For details, see *pfKeyEquate*.

## In a formGroup part

In a formGroup part, a use declaration refers to a form that is specified outside the form group. This kind of declaration allows multiple form groups to share the same form.

The syntax for a use declaration in a formGroup part is as follows:

```

▶▶ use formPartName ;

```

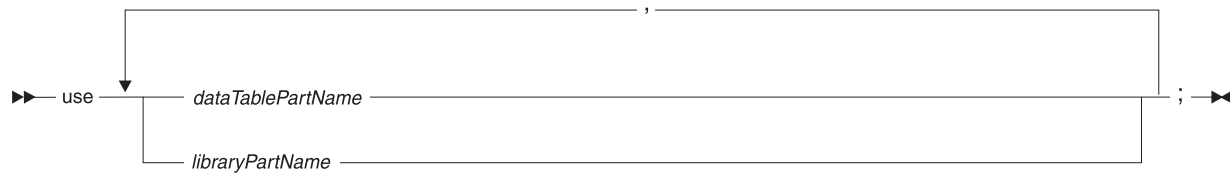
*formPartName*

Name of a form part that is visible to the form group. For an overview of forms, see *Form part*.

You cannot override properties of a form part in the use declaration of a formGroup part.

## In a pageHandler part

Each use declaration in a pageHandler part must be external to any function. The syntax for the declaration is as follows:



### *dataTablePartName*

Name of a dataTable part that is visible to the pageHandler part.

You cannot override properties of a dataTable part in the use declaration.

For an overview of dataTable parts, see *DataTable part*.

### *libraryPartName*

Name of a library part that is visible to the pageHandler part.

You cannot override properties of the library part in the use declaration.

For an overview of library parts, see *Library part*.

### **Related concepts**

"DataTable" on page 176

"FormGroup part" on page 183

"Form part" on page 184

"Library part of type basicLibrary" on page 169

"Library part of type basicLibrary" on page 169

"References to parts" on page 23

### **Related reference**

"pfKeyEquate" on page 792



---

## EGL Java runtime error codes

When an error occurs at Java run time, EGL places an error code in the system variable `sysVar.errorCode` and in most cases presents a message that has the same identifier as the error code. You can cause a customized message to be displayed in place of the EGL message; for details, see *Message customization for EGL Java run time*.

The error situations are as follows:

- A failure occurs during a remote call, an EJB call, a commit, or a rollback. In those cases, the message identifier begins with CSO.
- An error occurs in a Web application. In a subset of those cases, the message identifier begins with EGL.
- An error occurs during a local call, during access of a file or database, or during execution of one the following system functions--
  - Math functions
  - String functions
  - `sysLib.convert`In those cases, the message identifier begins with VGJ.
- An error occurs in a Java access function. In that case, the error code includes only numbers, and no message is displayed.

The error codes that are assigned by the Java access functions are shown in the next table. The other error codes are shown in the next sections.

Value in <code>sysVar.errorCode</code>	Description
00001000	An exception was thrown by an invoked method or as a result of a class initialization.
00001001	The object was null, or the specified identifier was not in the object space.
00001002	A public method, field, or class with the specified name does not exist or cannot be loaded.
00001003	The EGL primitive type does not match the type expected in Java.
00001004	The method returned null, the method does not return a value, or the value of a field was null.
00001005	The returned value does not match the type of the return item.
00001006	The class of an argument cast to null could not be loaded.
00001007	A <code>SecurityException</code> or <code>IllegalAccessException</code> was thrown during an attempt to get information about a method or field, or an attempt was made to set the value of a field that was declared final.
00001008	The constructor cannot be called; the class name refers to an interface or abstract class.
00001009	An identifier rather than a class name must be specified; the method or field is not static.

#### Related reference

"I/O error values" on page 638

"Message customization for EGL Java run time" on page 768

"errorCode" on page 1068

---

## EGL Java runtime error code CSO7000E

**CSO7000E: An entry for the specified called program %1 cannot be found in the linkage properties file %2.**

### Explanation

The message occurs in this situation:

- When the calling program was generated, property **remoteBind** was set to **RUNTIME** in the linkage options part, in the **callLink** element for the called program; and
- An entry for the specified called program cannot be found at run time, in the linkage properties file. The reason may be one of the following:
  - The linkage properties file cannot be found.
  - The file was found, but an entry for the called program is not in that file.
  - An incorrect linkage properties file was specified.

### User Response

Do as follows:

- If the program is being called from a Java wrapper, the linkage properties file must be named *link.properties*, where *link* is the name of the linkage options part used at generation. Make sure the file exists, has an entry for the called program, and is in a directory or archive specified in the CLASSPATH variable.
- If the program is being called from a program running in the J2EE environment, the linkage properties file can be identified by the `cso.linkageOptions.link` environment variable in the deployment descriptor, where *link* is the name of the linkage options part used at generation. If the environment variable is not set, the linkage properties file must be named *link.properties*, where *link* is the name of the linkage options part used at generation. Make sure the file exists, has an entry for the called program, and is in a directory or archive specified in CLASSPATH.
- If the program is being called from a program not running in the J2EE environment, the situation is as follows:
  - The linkage properties file can be identified by the `cso.linkageOptions.link` property, where *link* is the name of the linkage options part used at generation. If the property is not set, the linkage properties file may be named *link.properties*, where *link* is the name of the linkage options part used at generation. In these two cases, make sure the file exists, has an entry for the called program, and is in a directory or archive specified in CLASSPATH.
  - If the linkage properties file cannot be found, the linkage properties must be in the program properties file; in that case, make sure the program properties file includes an entry for the called program and that the program properties file is in a directory or archive specified in CLASSPATH.

For other details, see the EGL help pages on the **callLink** element, on Java runtime properties, and on setting up the environment.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
- The type of internal error

2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code CSO7015E

**CSO7015E: Cannot open the linkage properties file %1.**

### Explanation

The linkage properties file cannot be opened because the file is locked or cannot be found.

### User Response

Make sure that the linkage properties file is not locked by another process and that the file resides in a directory or archive specified in your CLASSPATH.

---

## EGL Java runtime error code CSO7016E

**CSO7016E: The properties file csouidpwd.properties ould not be read. Error: %1**

### Explanation

The file was found but there was an error reading from it.

### User Response

Use the Error portion of the message to diagnose and correct the problem.

---

## EGL Java runtime error code CSO7020E

**CSO7020E: The conversion table %1 is not valid.**

### Explanation

A conversion table that handles bidirectional text is invalid or cannot be loaded.

### User Response

The conversion table must reside in a directory or archive specified in the CLASSPATH. For details on developing the conversion table, see the help page on bidirectional text.

---

## EGL Java runtime error code CSO7021E

**CSO7021E: The client text attribute tag %2 in conversion table %1 is not valid.**

### Explanation

The conversion table file is not valid.

### User Response

Correct the file and run the program again.

---

## EGL Java runtime error code CSO7022E

**CSO7022E: The server text attribute tag %2 in conversion table %1 is not valid.**

### Explanation

The conversion table file is not valid.

### User Response

Correct the file and run the program again.

---

## EGL Java runtime error code CSO7023E

**CSO7023E: The value %3 for Arabic option tag %2 in conversion table %1 is not valid.**

### Explanation

The conversion table file is not valid.

### User Response

Correct the file and run the program again.

---

## EGL Java runtime error code CSO7024E

**CSO7024E: The value %3 for Wordbreak option tag %2 in conversion table %1 is not valid.**

### Explanation

The conversion table file is not valid.

### User Response

Correct the file and run the program again.

---

## EGL Java runtime error code CSO7026E

**CSO7026E: The value %3 for Roundtrip option tag %2 in conversion table %1 is not valid.**

### **Explanation**

The conversion table file is not valid.

### **User Response**

Correct the file and run the program again.

---

## **EGL Java runtime error code CSO7045E**

**CSO7045E: Error obtaining the address of entry point %1 within the shared library %2. RC = %3.**

### **Explanation**

An error was encountered in obtaining the address of the entry point within the shared library.

### **User Response**

Make sure that the referenced shared library is the correct shared library to be loaded. If so, make sure the shared library is built correctly.

---

## **EGL Java runtime error code CSO7050E**

**CSO7050E: An error occurred in remote program %1, date %2, time %3**

### **Explanation**

An error occurred in a called program, and the program stopped running.

### **User Response**

Use the date and time stamp on this message to associate the message with any diagnostic messages logged at the remote location. Check those diagnostic messages for further details.

---

## **EGL Java runtime error code CSO7060E**

**CSO7060E: An error was encountered while loading the shared library %1. The return code is %2.**

### **Explanation**

An error was encountered while loading the shared library.

### **User Response**

Make sure the shared library resides in a directory specified in your PATH or LIBPATH environment variable. Make sure that the shared library is built correctly.

---

## **EGL Java runtime error code CSO7080E**

**CSO7080E: The specified protocol %1 is not valid.**



### **Explanation**

The specified protocol in the linkage is unrecognized.

### **User Response**

Consult the documentation and specify a valid protocol.

---

## **EGL Java runtime error code CSO7160E**

**CSO7160E: An error occurred in remote program %1, date %2, time %3, on system %4.**

### **Explanation**

The Java program that you are running calls a remote program on the specified system, which failed in execution at the date and time specified.

### **User Response**

Check the remote server log for a more detailed description in problem analysis.

---

## **EGL Java runtime error code CSO7161E**

**CSO7161E: Run unit ended due to an application error on system %1 trying to call program %2. %3**

### **Explanation**

An error occurred at the remote server that causes the remote run unit to terminate abnormally when executing the remote program. Diagnostic messages preceding this message in the server job log explain the nature of error. If available, additional information may be included with the message text.

### **User Response**

Check the error messages logged on the server system to determine what to do to fix the original problem.

---

## **EGL Java runtime error code CSO7162E**

**CSO7162E: Invalid password or user ID supplied for connecting to system %1. Java exception message received: %2.**

### **Explanation**

The password or user ID supplied to connect to the remote system is not set or not valid.

### **User Response**

Verify that the connection is set. Verify that the user ID and password supplied to the remote system are correct, and try again.

---

## EGL Java runtime error code CSO7163E

**CSO7163E: Remote access security error to system %1 for user %2. Java exception message received: %3**

### Explanation

The specified user currently connecting to the system does not have sufficient authority or does not have access to the remote resource on the specified system.

### User Response

Verify that the user connecting to the remote machine has the proper authority to connect to the remote machine and to execute the remote server program.

---

## EGL Java runtime error code CSO7164E

**CSO7164E: Remote connection error to system %1. Java exception message received: %2**

### Explanation

An error occurred when communicating or connecting to the remote system.

### User Response

Check that the remote server is available; then retry. If this does not work, contact the remote host's system administrator to determine the actual problem.

---

## EGL Java runtime error code CSO7165E

**CSO7165E: Commit failed on system %1. %2**

### Explanation

A commit operation failed on the remote system.

### User Response

Diagnose the problem by reviewing the detailed message, which is shown here as %2.

---

## EGL Java runtime error code CSO7166E

**CSO7166E: Rollback failed on system %1. %2**

### Explanation

A rollback operation failed on the remote system.

### User Response

Diagnose the problem by reviewing the detailed message, which is shown here as %2.

---

## EGL Java runtime error code CSO7360E

**CSO7360E: AS400Toolbox execution error: %1, %2 while calling program %3 on system %4**

### Explanation

The Java program or applet that you are running uses the Java400 protocol to call a remote server program. An unexpected exception was caught while attempting to call the server program. The message text consists of the name of the AS400 Toolbox exception followed by the message returned with the exception.

### User Response

Use the AS400 Toolbox error message provided to analyze the cause of the problem.

---

## EGL Java runtime error code CSO7361E

**CSO7361E: EGL OS/400 Host Services error. Required files not found on system %1.**

### Explanation

The Java program or applet that you are running uses the Java400 protocol to call a remote server program. An exception is raised when the remote catcher is not found or is not in the proper library on the server.

### User Response

Check that EGL OS/400 Host Services is properly installed on the remote system. Apply the latest PTFs if available.

---

## EGL Java runtime error code CSO7488E

**CSO7488E: Unknown TCP/IP hostname: %1**

### Explanation

An UnknownHostException was thrown during an attempt to connect to the remote TCP/IP listener program.

### User Response

Do as follows:

- Add the property `cso.serverLinkage.xxx.location` to the runtime linkage properties file, where `xxx` is the name of the called program or is an application name, as described in the EGL reference-type help page on the linkage properties file. The value of the property is a valid TCP/IP host name.
- Alternatively, set the TCP/IP host name at generation time and regenerate the program:
  - In the linkage options part, in the `callLink` element for the called program, set property **location** to the TCP/IP host name

- If you wish to finalize linkage options only at run time, set property **remoteBind** to **RUNTIME** and generate with build descriptor option **genProperties** set to **YES**

For other details, see the EGL help pages on the callLink element, on the linkage properties file, and on setting up the environment.

---

## EGL Java runtime error code CSO7489E

**CSO7489E: The linkage information used to call the program is inconsistent or missing.**

### Explanation

The program was unable to determine how the program should be called.

### User Response

Supply all required linkage information. The information that is required depends on the desired type of call. Refer to the help pages on the linkage options part, particularly on the callLink element.

---

## EGL Java runtime error code CSO7610E

**CSO7610E: An error was encountered while calling CICS ECI to commit a unit of work. The CICS return code is %1.**

### Explanation

A commit request was issued by the client but was not successful. An error was encountered while calling CICS External Call Interface to commit a logical unit of work.

### User Response

Please refer to the appropriate CICS documentation for the corrective actions for the specified error.

---

## EGL Java runtime error code CSO7620E

**CSO7620E: An error was encountered while calling the CICS ECI to rollback a unit of work. The CICS return code is %1.**

### Explanation

A rollback request was issued by the client but was not successful. An error was encountered while calling CICS External Call Interface to rollback a logical unit of work.

### User Response

Please refer to the appropriate CICS documentation for the corrective actions for the specified error.

---

## EGL Java runtime error code CSO7630E

**CSO7630E: An error was encountered while ending the remote procedure call to a CICS server. The CICS return code is %1.**

### Explanation

An attempt was made to commit all open logical units of work before ending the EGL remote procedure call to a CICS server but was not successful. This request was made via the CICS External Call Interface.

### User Response

Refer to the appropriate CICS documentation for the corrective actions for the specified error.

---

## EGL Java runtime error code CSO7640E

**CSO7640E: %1 is an invalid value for the ctgport entry.**

### Explanation

The value of ctgport must be an integer.

### User Response

Use the correct ctgport number.

---

## EGL Java runtime error code CSO7650E

**CSO7650E: An error was encountered calling program %1 using the CICS ECI. Return code: %2. CICS system identifier: %3.**

### Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

### User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7651E

**CSO7651E: An error was encountered calling program %1 using the CICS ECI. Return code: -3 (ECI\_ERR\_NO\_CICS). CICS system identifier: %2.**

### Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -3 - ECI\_ERR\_NO\_CICS  
Client or server system not available

### User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7652E

**CSO7652E: An error was encountered calling program %1 using the CICS ECI. Return code: -4 (ECI\_ERR\_CICS\_DIED). CICS system identifier: %2.**

### Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -4 - ECI\_ERR\_CICS\_DIED  
Server system no longer available

### User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7653E

**CSO7653E: An error was encountered calling program %1 using the CICS ECI. Return code: -6 (ECI\_ERR\_RESPONSE\_TIMEOUT). CICS system identifier: %2.**

### Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -6 - ECI\_ERR\_RESPONSE\_TIMEOUT  
Response time out. Time limit is specified in environment variable CSOTIMEOUT.

### User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7654E

**CSO7654E: An error was encountered calling program %1 using the CICS ECI. Return code: -7 (ECI\_ERR\_TRANSACTION\_ABEND). CICS system identifier: %2. Abend code: %3.**

### Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -7 - ECI\_ERR\_TRANSACTION\_ABEND  
Abnormal termination on server. Common ABEND codes are:
  - AEI0 - Server program not defined
  - AEI1 - Server transaction not defined

## User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7655E

**CSO7655E: An error was encountered calling program %1 using the CICS ECI. Return code: -22 (ECI\_ERR\_UNKNOWN\_SERVER). CICS system identifier: %2.**

### Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -22 - ECI\_ERR\_UNKNOWN\_SERVER  
Server system not defined

## User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7656E

**CSO7656E: An error was encountered calling program %1 using the CICS ECI. Return code: -27 (ECI\_ERR\_SECURITY\_ERROR). CICS system identifier: %2.**



## Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -27 - ECI\_ERR\_SECURITY\_ERROR  
User ID or password not valid

## User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7657E

**CSO7657E: An error was encountered calling program %1 using the CICS ECI. Return code: -28 (ECI\_ERR\_MAX\_SYSTEMS). CICS system identifier: %2.**

## Explanation

An error was returned from a CICS External Call Interface (ECI) function call when attempting to call a remote server program.

The system identifier is the name of the CICS system where the server program was to run. If blank, the system is specified in the CICS program definition for the program or in the CICS client initialization file. The return code is the CICS return code.

The CICS return code has the following meaning:

- -28 - ECI\_ERR\_MAX\_SYSTEMS  
Maximum number of servers reached

## User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code or if the return code is not documented above, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7658E

**CSO7658E: An error was encountered calling program %1 on system %2 for user %3. CICS ECI call returned RC %4 and Abend Code %5.**

### Explanation

A non-zero return code was returned on a CICS ECI call made from the gateway to the specified system on behalf of the user identified in the message.

### User Response

Correct the problem indicated by the return code.

For a complete explanation of the return code, refer to the CICS ECI documentation for your system for information on the corrective actions.

Return code values are associated with symbols in the CICS ECI include files faaecih.h or cics\_eci.h.

---

## EGL Java runtime error code CSO7659E

**CSO7659E: An exception occurred on the flow of an ECI Request to CICS system %1. Exception: %2**

### Explanation

An unexpected exception occurred in the flow method when attempting to send the ECI Request from the gateway to the CICS system identified in the message.

### User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO7669E

**CSO7669E: An error was encountered when connecting to CTG. CTG Location: %1, CTG Port: %2. Exception: %3**

### Explanation

An unexpected exception occurred when connecting to the CICS Transaction Gateway.

### User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO7670E

**CSO7670E: An error was encountered when disconnecting from CTG. CTG Location: %1, CTG Port: %2. Exception: %3**

### Explanation

An unexpected exception occurred when disconnecting from the CICS Transaction Gateway.

### User Response

Examine the exception string that was returned. If you are unable to determine the cause of the problem from the exception, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO7671E

**CSO7671E: When using CICSSSL protocol, both ctgKeyStore and ctgKeyStorePassword must be specified.**

### Explanation

Required values were not specified so the call cannot be completed.

### User Response

Make sure that both ctgKeyStore and ctgKeyStorePassword are specified.

---

## EGL Java runtime error code CSO7816E

**CSO7816E: A socket exception occurred when the gateway attempted to connect to server with hostname %1 and port %2 for userid %4. Exception was: %3**

### Explanation

The socket call to create and connect a socket from the gateway to the server system identified in the message failed with the exception shown.

The EGL gateway attempted a socket call to create and connect a TCP/IP socket for a server call. The socket call failed with the exception indicated in the message.

### User Response

Examine the exception information to determine a reason why a socket call from the gateway failed. If you are unable to determine the cause of the problem by examining the exception information, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO7819E

**CSO7819E: An unexpected exception occurred on function %2. Exception: %1**

## Explanation

The EGL gateway received an unexpected exception from the function identified in the message. An internal error may have occurred.

## User Response

If you are unable to determine the source of the problem from examining the exception information, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO7831E

**CSO7831E: The client's buffer was too small for the amount of data being passed on the call. Ensure that the cumulative size of the parameters being passed does not exceed the maximum allowed which is 32567 bytes.**

## Explanation

The buffer established by the client cannot be made as large as the cumulative size of the parameters being passed to the remote called program.

## User Response

Ensure that the cumulative size of the parameters being passed does not exceed the maximum allowed which is 32567 bytes. If they do not exceed the maximum and this error occurs, please report the error to IBM Support Center.

---

## EGL Java runtime error code CSO7836E

**CSO7836E: The client has received notification that the server is unable to start the remote called program. Reason code: %1.**

## Explanation

The server is unable to run the remote called program and has returned a reason code for problem determination.

## User Response

Reason codes are as follows:

- 2 - Server was unable to load the class for the called program. The server trace file may show more specific information. Make sure that the class is available to the server.

This problem may result from improper conversion of the class name passed to the server. Review the help page on data conversion to verify that the correct conversion table was specified in the linkage options part, in the callLink element for the called program, in property conversionTable.

- 3 - The called program was ended because of an error. The server trace file may show more specific information.

For any reason code not listed above or if you are unable to determine the cause of the failure, contact IBM support.

---

## EGL Java runtime error code CSO7840E

**CSO7840E: The client received notification from the server that the remote called program failed with return code %1.**

### Explanation

The remote called program ran but ended with a non-zero return code. The problem is in the program rather than in communications.

### User Response

Examine or trace the called program to determine why it completed with a non-zero return code.

---

## EGL Java runtime error code CSO7885E

**CSO7885E: A TCP/IP read function failed on a call for userid %2 to hostname %1. Exception returned was: %3**

### Explanation

The EGL gateway received an exception when attempting a TCP/IP read function.

### User Response

Examine the exception information returned in order to determine the cause of the problem. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO7886E

**CSO7886E: A TCP/IP write function failed on a call for userid %2 to hostname %1. Exception returned was: %3**

### Explanation

The EGL gateway received an exception when attempting a TCP/IP write function.

### User Response

Examine the exception information returned in order to determine the cause of the problem. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO7955E

**CSO7955E: %1, %2**

### Explanation

An unexpected Java exception was caught.

The message text shows the name of the Java exception followed by the Java message that was thrown with the exception.

## User Response

Review the message and respond as appropriate.

---

### EGL Java runtime error code CSO7957E

**CSO7957E: Conversion table name %1 is not valid for Java data conversion.**

#### Explanation

You are using a generated Java class to call a program and have incorrectly specified a conversion table to convert Java data to the format used by the called program.

#### User Response

Review the help page on data conversion to determine the conversion table name, which you specify in the linkage options part, in the callLink element for the called program, in property conversionTable.

---

### EGL Java runtime error code CSO7958E

**CSO7958E: The native code did not provide an object of type CSOPowerServer to the Java wrapper, as is needed to convert data between the Java wrapper and the EGL-generated program.**

#### Explanation

The native Java code invoked the call or execute method of a Java wrapper without first instantiating an object of class CSOPowerServer and providing that object to the wrapper.

#### User Response

Review the help pages on the Java wrapper for details on accessing EGL middleware, as is always required for data conversion.

---

### EGL Java runtime error code CSO7966E

**CSO7966E: The code page encoding %1 was not found for the conversion table %2.**

#### Explanation

The conversion table specified in the linkage options requires an encoding not available in the Java Virtual Machine (JVM) being used.

#### User Response

Review the help page on data conversion to determine the correct conversion table name, which you specify in the linkage options part, in the callLink element for the called program, in property conversionTable. If you specified the correct conversion table, make sure that the JVM that you are using is supported by the Java runtime environment of EGL.

If the previous steps do not reveal the problem, consider whether the installation of your JVM is flawed or whether your Java Virtual machine does not support all encodings. In these cases, refer to the documentation of your JVM vendor or contact the JVM vendor for assistance.

If you encountered the error when running an applet client in a browser, the error occurred at the PowerServer SessionManager used by the client applet. In this case, refer to the documentation for the JVM that the SessionManager is running on or contact the JVM vendor.

---

## EGL Java runtime error code CSO7968E

**CSO7968E: Host %1 is not known or could not be found.**

### Explanation

No remote system specified in the linkage.

### User Response

The remote system must be specified in the linkage part's location field.

---

## EGL Java runtime error code CSO7970E

**CSO7970E: Could not load the required EGL shared library %1, reason: %2**

### Explanation

The shared library for is required to complete the operation, but it could not be loaded.

### User Response

Make sure that the shared library is on the system. It must be included in the environment variable that specifies the shared library path, PATH or LIBPATH.

---

## EGL Java runtime error code CSO7975E

**CSO7975E: The properties file %1 could not be opened.**

### Explanation

The properties file required by the program could not be opened. The name of the properties file may be specified on the command line when the program is started. If no name is given when the program is started, the following name is used by default:

`tcpiplistener.properties`

Either the properties file does not exist, or it exists but could not be opened.

### User Response

Ensure that the properties file exists and that the program has the proper permissions to read it, then run the program again.

---

## EGL Java runtime error code CSO7976E

**CSO7976E: The trace file %1 could not be opened. The exception is %2 The message is as follows: %3**

### Explanation

An exception occurred when the program tried to open the trace output file.

### User Response

Correct the problem and re-run the program.

---

## EGL Java runtime error code CSO7977E

**CSO7977E: The program properties file does not contain a valid setting for the %1 property, which is required.**

### Explanation

The property is not defined in the program properties file.

### User Response

Add the property to the program properties file and re-run the program. For details, see the help page on Java runtime properties.

---

## EGL Java runtime error code CSO7978E

**CSO7978E: An unexpected exception occurred. The exception is %1 The message is as follows: %2**

### Explanation

The program encountered an error.

### User Response

Correct the problem and re-run the program.

---

## EGL Java runtime error code CSO7979E

**CSO7979E: Unable to create an InitialContext. Exception is %1**

### Explanation

The exception was thrown from the constructor of `javax.naming.InitialContext`. The program needs to create the `InitialContext` object to access the J2EE environment settings.

### User Response

Use the text of the exception and the documentation of your J2EE environment to correct the problem.



---

## EGL Java runtime error code CSO8000E

**CSO8000E: The password entered to the Gateway has expired.  
%1**

### Explanation

The EGL GatewayServlet received an expired password exception when attempting to authenticate the user with the provided password.

### User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by proving a new password.

---

## EGL Java runtime error code CSO8001E

**CSO8001E: The password entered to the Gateway is not valid.  
%1**

### Explanation

The EGL GatewayServlet received an invalid password exception when attempting to authenticate the user with the provided password.

### User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by proving a new password.

---

## EGL Java runtime error code CSO8002E

**CSO8002E: The userid entered to the Gateway is not valid. %1**

### Explanation

The EGL GatewayServlet received an invalid userid exception when attempting to authenticate the user with the provided userid.

### User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by proving a new userid.

---

## EGL Java runtime error code CSO8003E

**CSO8003E: Null entry for %1**

### Explanation

Null entry has been detected.

### User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing required entry.

---

## EGL Java runtime error code CSO8004E

**CSO8004E: The gateway received an unknown security error.**

### Explanation

The EGL GatewayServlet received an unknown security exception when attempting to authenticate the user with the provided user information.

### User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing new user information. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO8005E

**CSO8005E: Error occurred when changing the password. %1**

### Explanation

The EGL GatewayServlet received an error when attempting to change the provided password.

### User Response

Examine the exception information returned in order to determine the cause of the problem. Correct the problem by providing new password. If you are unable to determine why the failure occurred, please contact IBM Support for assistance.

---

## EGL Java runtime error code CSO8100E

**CSO8100E: Unable to get a connection factory. Exception is %1**

### Explanation

The exception was thrown during a look-up of the connection factory that is used on a call when the value of the remoteComType property is CICSJ2C, IMSJ2C, or IMSTCP. The remoteComType property is in the linkage options part, in the callLink element for the called program.

For CICSJ2C and IMSJ2C the name of the connection factory is determined by the resource adapter installed on the application server used to run the calling program. The JNDI name of the resource adapter being used is the value that you set in the location property of the same callLink element.

For IMSTCP the connection factory is the connection factory required by IMS Connector for Java.

### User Response

For CICSJ2C and IMSJ2C make sure that the resource adapter is defined properly in the J2EE environment and that the value for the location property is correct in the callLink element for the called program.

For IMSTCP, make sure that the IMS Connector for Java jar files are included in the classpath when the calling application is started.

---

## EGL Java runtime error code CSO8101E

**CSO8101E: Unable to get a connection. Exception is: %1**

### Explanation

The exception was thrown by the getConnection method of the ConnectionFactory object that was used to make a call when the value of the remoteComType property is CICSJ2C, IMSJ2C, or IMSTCP. The remoteComType property is in the linkage options part, in the callLink element for the called program.

### User Response

For CICSJ2C and IMSJ2C the Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

For IMSTCP, make sure that the IMS Connector for Java jar files are included in the classpath when the calling application is started. Diagnose the problem by using the text of the exception and IMS Connector for Java documentation

---

## EGL Java runtime error code CSO8102E

**CSO8102E: Unable to get an Interaction. Exception is: %1**

### Explanation

The exception was thrown by the createInteraction method of the Connection object that is used to make a call when the value of the remoteComType property is CICSJ2C, IMSJ2C, or IMSTCP. The remoteComType property is in the linkage options part, in the callLink element for the called program.

### User Response

For CICSJ2C and IMSJ2C the Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

For IMSTCP, make sure that the IMS Connector for Java jar files are included in the classpath when the calling application is started. Diagnose the problem by using the text of the exception and IMS Connector for Java documentation.

---

## EGL Java runtime error code CSO8103E

**CSO8103E: Unable to set an interaction verb. Exception is %1**

### Explanation

The exception was thrown by the setInteractionVerb method of the ECIInteractionSpec or IMSInteractionSpec object that is used to make a call when the value of the remoteComType property is CICSJ2C or IMSTCP, respectively. The

remoteComType property is in the linkage options part, in the callLink element for the called program.

### User Response

For CICSJ2C the Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

For IMSTCP, make sure that the IMS Connector for Java jar files are included in the classpath when the calling application is started. Diagnose the problem by using the text of the exception and IMS Connector for Java documentation.

---

## EGL Java runtime error code CSO8104E

**CSO8104E: An error occurred during an attempt to communicate with CICS. Exception is %1**

### Explanation

The exception was thrown by the execute method of the Interaction object that is used to make a call when the value of the remoteComType property is CICSJ2C. The remoteComType property is in the linkage options part, in the callLink element for the called program.

### User Response

The Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment. Additional information may be in the CICS Transaction Gateway log or in a log file on the remote system.

---

## EGL Java runtime error code CSO8105E

**CSO8105E: Unable to close an Interaction or Connection. Exception is %1**

### Explanation

The exception was thrown by the close method of a Connection or Interaction object that is used to make a call when the value of the remoteComType property is CICSJ2C, IMSJ2C, or IMSTCP. The remoteComType property is in the linkage options part, in the callLink element for the called program.

### User Response

For CICSJ2C and IMSJ2C the Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

For IMSTCP, make sure that the IMS Connector for Java jar files are included in the classpath when the calling application is started. Diagnose the problem by using the text of the exception and the documentation of IMS Connector for Java

---

## EGL Java runtime error code CSO8106E

**CSO8106E: Unable to get a LocalTransaction for client unit of work. Exception is %1**

### Explanation

The exception was thrown by the `getLocalTransaction` method of a `Connection` object that is used to make a call in this situation:

- The value of the `remoteComType` property is `CICSJ2C`
- The value of the `luwControl` property is `CLIENT`

Those properties are in the linkage options part, in the `callLink` element for the called program.

### User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

---

## EGL Java runtime error code CSO8107E

**CSO8107E: Unable to set the timeout value on a CICSJ2C call. Exception is %1**

### Explanation

The exception was thrown by the `setExecuteTimeout` method of an `ECIInteractionSpec` object that is used to make a call when the value of the `remoteComType` property is `CICSJ2C`. The `remoteComType` property is in the linkage options part, in the `callLink` element for the called program.

### User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

---

## EGL Java runtime error code CSO8108E

**CSO8108E: An error occurred during an attempt to communicate with CICS.**

### Explanation

The `execute` method of the `Interaction` object that is used to make the call returned false. The call did not complete successfully.

### User Response

The Connection Factory or Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the

documentation of your Resource Adapter, and the documentation of your J2EE environment. Additional information may be in the gateway log or in a log file on the remote system.

---

## EGL Java runtime error code CSO8109E

**CSO8109E: The timeout value %1 is invalid. It must be a number.**

### Explanation

An invalid value was specified for the timeout.

### User Response

Either do not specify a timeout value, or specify a number.

---

## EGL Java runtime error code CSO8110E

**CSO8110E: The parmForm linkage property must be set to COMMPTR to call program %1 as there is at least one parameter that is a dynamic array.**

### Explanation

The parmForm must be COMMPTR because one of the parameters is a dynamic array.

### User Response

Change the parmForm to COMMPTR.

---

## EGL Java runtime error code CSO8115E

**CSO8115E: An error occurred during an attempt to communicate with IMS. Exception is %1**

### Explanation

The exception was thrown by the execute method of the Interaction object that is used to make a call when the value of the remoteComType property is IMSJ2C or IMSTCP. The remoteComType property is in the linkage options part, in the callLink element for the called program.

### User Response

For IMSJ2C the Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

For IMSTCP, make sure that the IMS Connector for Java jar files are included in the classpath when the calling application is started. Diagnose the problem by using the text of the exception and the documentation of IMS Connector for Java

Additional information may be in an IMS Connect for Java log where the calling program is run, or in a log file on the remote system.

---

## EGL Java runtime error code CSO8117E

**CSO8117E: An error occurred during an attempt to communicate with IMS**

### Explanation

The execute method of the Interaction object that is used to make the call returned false. The failure occurred in the execute method of the Interaction object that is used to make a call when the value of the remoteComType property is IMSJ2C or IMSTCP. The remoteComType property is in the linkage options part, in the callLink element for the called program. The call did not complete successfully but no exception was thrown.

### User Response

For IMSJ2C the IMS Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

For IMSTCP, make sure that the IMS Connector for Java jar files are included in the classpath when the calling application is started. Diagnose the problem by using the text of the exception and the documentation of IMS Connector for Java

Additional information may be found in an IMS Connector for Java log or in an IMS Connect log file on the remote system.

---

## EGL Java runtime error code CSO8180E

**CSO8180E: The linkage specified a DEBUG call within a J2EE server. The call was not made on a J2EE server, the J2EE server is not in debug mode, or the J2EE server was not enabled for EGL debugging.**

### Explanation

The DEBUG call cannot be completed.

### User Response

If the call is not being made on a J2EE server, the TCP/IP hostname of the machine running the EGL debugger must be specified in the location field of the linkage. If the call is being made on a J2EE server, make sure that it was started in debug mode and make sure that the EGL Debugger jar files were added to it.

---

## EGL Java runtime error code CSO8181E

**CSO8181E: Cannot contact the EGL debugger at hostname %1 and port %2. Exception is %3**

### Explanation

The DEBUG call cannot be completed because the EGL debugger could not be contacted.

## User Response

Make sure an EGL Listener is running in the EGL debugger at the specified hostname and port.

---

## EGL Java runtime error code CSO8182E

**CSO8182E: An error occurred while communicating with the EGL debugger at hostname %1 and port %2. Exception is %3**

### Explanation

Communication between the EGL debugger and the calling program failed.

### User Response

Use the information in the exception message to correct the problem.

---

## EGL Java runtime error code CSO8200E

**CSO8200E: Array wrapper %1 cannot be expanded beyond its maximum size. The error occurred in method %2.**

### Explanation

The maximum size of the array was exceeded.

### User Response

Check the size and maximum size of the array before attempting to add to it.

---

## EGL Java runtime error code CSO8201E

**CSO8201E: %1 is an invalid index for array wrapper %2. Maximum size: %3. Current® size: %4**

### Explanation

The index is outside the bounds of the array.

### User Response

Use a valid index.

---

## EGL Java runtime error code CSO8202E

**CSO8202E: %1 is not a valid maximum size for array wrapper %2.**

### Explanation

The property maxSize must be greater than or equal to zero.

### User Response

Do not set the property maxSize to a negative number.



---

## EGL Java runtime error code CSO8203E

**CSO8203E: %1 is an invalid object type to add to an array wrapper of type %2.**

### Explanation

The contents of the array must match its definition.

### User Response

Change the type of objects that the array stores, or do not attempt to store that type of object in the array.

---

## EGL Java runtime error code CSO8204E

**CSO8204E: Cannot pass an Any, Dictionary, ArrayDictionary, Blob, Clob, or Ref variable as a parameter.**

### Explanation

The types listed may not be used as parameters on a call statement. In addition, types that contain the listed types may not be used as parameters.

### User Response

Do not pass that kind of parameter to the called program.

---

## EGL Java runtime error code CSO8117E

**CSO8117E: An error occurred during an attempt to communicate with IMS**

### Explanation

The execute method of the Interaction object that is used to make the call returned false. The failure occurred in the execute method of the Interaction object that is used to make a call when the value of the remoteComType property is IMSJ2C or IMSTCP. The remoteComType property is in the linkage options part, in the callLink element for the called program. The call did not complete successfully but no exception was thrown.

### User Response

For IMSJ2C the IMS Resource Adapter may not be defined or configured properly. Diagnose the problem by using the text of the exception, the documentation of your Resource Adapter, and the documentation of your J2EE environment.

For IMSTCP, make sure that the IMS Connector for Java jar files are included in the classpath when the calling application is started. Diagnose the problem by using the text of the exception and the documentation of IMS Connector for Java

Additional information may be found in an IMS Connector for Java log or in an IMS Connect log file on the remote system.

---

## EGL Java runtime error code EGL0650E

**EGL0650E: The %1RequestAttr function failed with key, %2.  
Error: %3**

### Explanation

The EGL GetRequestAttr or SetRequestAttr function failed when invoked with the given key.

### User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the function is used within a PageHandler function.

---

## EGL Java runtime error code EGL0651E

**EGL0651E: The %1SessionAttr function failed with key, %2.  
Error: %3**

### Explanation

The EGL GetSessionAttr or SetSessionAttr function failed when invoked with the given key.

### User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the function is invoked within a PageHandler function.

---

## EGL Java runtime error code EGL0652E

**EGL0652E: The forward statement failed with label, %1. Error: %2**

### Explanation

Control could not be forwarded to the given label.

### User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the EGL object which associated with the label is generated correctly and that the label is defined in the application configuration file.

---

## EGL Java runtime error code EGL0653E

**EGL0653E: Failed to create Bean from EGL object, %1. Error: %2**

### Explanation

Could not create an access bean from EGL record or PageHandler definition.

### User Response

Use the Error part of this message to diagnose and correct the problem.

---

## EGL Java runtime error code EGL0654E

**EGL0654E: The SetError function failed with item, %1, key, %2.  
Error: %3**

### Explanation

The SetError function failed when invoked with the given message key.

### User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the item has an error entry in the JSP and the key is defined in the message resource file.

---

## EGL Java runtime error code EGL0655E

**EGL0655E: Failed to copy data from Bean to EGL record, %1.  
Error: %2**

### Explanation

An attempt to move data from the form bean to the record failed.

### User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the bean definition matches with the record definition.

---

## EGL Java runtime error code EGL0656E

**EGL0656E: Cannot assign array of size %1 to static array of size %2.**

### Explanation

The sizes of the arrays must match.

### User Response

Check the EGL array definitions and make sure the array sizes are the same.

---

## EGL Java runtime error code EGL0657E

**EGL0657E: Processing of an onPageLoad parameter failed.  
Error: %1.**

### Explanation

An error occurred when EGL tried to receive values into the parameters of the onPageLoad function.

## User Response

Use the Error part of this message to diagnose and correct the problem. Make sure the type definition of the passed value matches the type defined for the parameter in the `onPageLoad` function.

---

## EGL Java runtime error code VGJ0001E

**VGJ0001E: Maximum value overflow from %1.**

### Explanation

During an arithmetic calculation, either a value was divided by zero or an intermediate result exceeded 18 significant digits. The program ends unless system variable `VGVar.handleOverflow` is set to 2.

### User Response

Perform one or more of the following actions:

- Correct the logic of your program to avoid the error.
- Define the program logic to handle the overflow condition; use the system variables `VGVar.handleOverflow` and `overflowIndicator`.

---

## EGL Java runtime error code VGJ0002E

**VGJ0002E: Error %1 occurred. The message text for this error could not be found in the message file %2.**

### Explanation

The message file may be corrupt or from an older release of EGL.

### User Response

Complete one of the following instructions:

- If you extracted class files from the file `fda6.jar`, verify that the classes you have are at the same release or maintenance level as the classes in that file. If you find a mismatch, replace the older classes with the correct version.
- Reinstall `fda6.jar` from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the product's installation manual.

---

## EGL Java runtime error code VGJ0003E

**VGJ0003E: An internal error occurred at location %1.**

## Explanation

This error can occur only when system constraints or requirements were not satisfied or when EGL program parts were used improperly. The location specified in the error is used only for IBM diagnostic purposes.

## User Response

Check the program setup and restart the system. If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
- The type of internal error

2. Record the situation in which this message occurs.
3. For further instructions on how to report possible defects to the IBM Support Center, refer to the product's installation manual.

---

## EGL Java runtime error code VGJ0004I

**VGJ0004I: The error occurred in %1, function %2.**

## Explanation

This message accompanies another message when an error occurs. It identifies the program or record where the error occurred, as well as the function that was executing at the time.

## User Response

None.

---

## EGL Java runtime error code VGJ0005I

**VGJ0005I: The error occurred in %1.**

## Explanation

This message accompanies another message and identifies the program or record where an error occurred.

## User Response

None.

---

## EGL Java runtime error code VGJ0006E

**VGJ0006E: An error occurred during an I/O operation. %1**

## Explanation

An I/O operation failed, and the EGL statement has no try statement to deal with the error.

## User Response

If you want the program to handle the error, set `handleHardIOErrors` to 1 and put the I/O statement in a try statement, as in the following example:

```
VGVar.handleHardIOErrors = 1;

if (userRequest == "A")
  try
    add record1
  onException
    myErrorHandler(12);
  end
end
```

---

## EGL Java runtime error code VGJ0007E

**VGJ0007E: Minimum value overflow from %1.**

### Explanation

The arithmetic operation has produced a result that is beyond the minimum value allowed for the data type.

### User Response

Adjust the arithmetic expression accordingly.

---

## EGL Java runtime error code VGJ0008E

**VGJ0008E: A recoverable resource error occurred. %1**

### Explanation

There was an error while closing, committing, or rolling back a recoverable resource.

### User Response

Use the information in the error message to correct the problem.

---

## EGL Java runtime error code VGJ0009E

**VGJ0009E: No field with identifier %1 could be found in %2.**

### Explanation

A dynamic access failed because the specified field does not exist.

### User Response

Do not access nonexistent fields.

---

## EGL Java runtime error code VGJ0010E

**VGJ0010E: The assignment to %1 failed: incompatible assignment source %2.**

**Explanation**

The source's type is not one that may be assigned to the target.

**User Response**

Ensure that the source and target types are compatible when assigning values.

---

**EGL Java runtime error code VGJ0011E**

**VGJ0011E: Cannot resolve the value of %1 to a primitive type.**

**Explanation**

The variable was used as a data item, but it is not a data item.

**User Response**

Change the program so that it does not use the variable as if it were a data item.

---

**EGL Java runtime error code VGJ0012E**

**VGJ0012E: Could not evaluate an arithmetic expression: incompatible types in %1.**

**Explanation**

The types of the values in the expression are incompatible.

**User Response**

Change the program to use compatible types in the expression.

---

**EGL Java runtime error code VGJ0013E**

**VGJ0013E: The set statement failed: %1 cannot be set to the %2 state.**

**Explanation**

The specified state is not supported for the variable.

**User Response**

Change the program so that it does not attempt this operation.

---

**EGL Java runtime error code VGJ0014E**

**VGJ0014E: %1 cannot be subscripted. It is not an array.**

**Explanation**

The variable was used as an array, but it is not an array.

### **User Response**

Change the program so that it does not use the variable as an array.

---

## **EGL Java runtime error code VGJ0015E**

**VGJ0015E: %1, %2**

### **Explanation**

There was an error. The exception and its message are used as inserts to this message.

### **User Response**

Use the information from the message inserts to correct the problem.

---

## **EGL Java runtime error code VGJ0016E**

**VGJ0016E: Any variable %1 has not been given a value.**

### **Explanation**

The variable was used before having been assigned a value.

### **User Response**

Change the program to assign a value to the variable before using it.

---

## **EGL Java runtime error code VGJ0017E**

**VGJ0017E: A reference variable was used, but its value is Nil.**

### **Explanation**

The variable must reference a non-Nil value before it can be used.

### **User Response**

Before using the variable, give it a value other than Nil.

---

## **EGL Java runtime error code VGJ0018E**

**VGJ0018E: Cannot perform dynamic access on structured record %1.**

### **Explanation**

Dynamic access is not permitted on a structured record.

### **User Response**

Do not use dynamic access on the structured record.



---

## EGL Java runtime error code VGJ0019E

**VGJ0019E: %1 cannot be copied.**

### Explanation

An operation attempted to copy something that may not be copied, or the attempt to make a copy failed.

### User Response

---

## EGL Java runtime error code VGJ0020E

**VGJ0020E: The variable named %1 cannot be used as a %2.**

### Explanation

The variable's type does not allow it to be used as if it were of the specified type.

### User Response

Change the program so that it does not use the variable as if it were a different type.

---

## EGL Java runtime error code VGJ0021E

**VGJ0021E: %1 cannot be tested for the %2 state.**

### Explanation

The error occurred in an IS or NOT expression. The variable on the left side of the expression does not support the state that was specified as the right side of the expression.

### User Response

Remove or modify the expression.

---

## EGL Java runtime error code VGJ0022E

**VGJ0022E: A timeout occurred in program %1 due to inactivity.**

### Explanation

A timeout occurred due to inactivity during a web transaction converse.

### User Response

Check for network errors and retry the process. Check for abnormal program termination. If this occurred check the program's error message.

---

## EGL Java runtime error code VGJ0023E

**VGJ0023E: Error loading data from reference type parameter %1.**

### **Explanation**

A function parameter has been defined as a reference type and was accessed as a primitive type.

### **User Response**

Change the code and remove the access to the reference type parameter.

---

## **EGL Java runtime error code VGJ0050E**

**VGJ0050E: An exception occurred while loading program %1.  
Exception: %2 Message: %3**

### **Explanation**

The program's class could not be loaded.

### **User Response**

Use the exception message to diagnose and fix the problem. The most common cause of this error is that the jar file or the directory containing the program's class file is not listed in the CLASSPATH environment variable.

---

## **EGL Java runtime error code VGJ0051E**

**VGJ0051E: Program %1 is not a web transaction.**

### **Explanation**

The program is not a web transaction, so it cannot be used as one. Either a User Interface record was passed on a transfer to a main or called program, or the gateway servlet sent a request to start a main or called program.

### **User Response**

Correct the program logic, or regenerate the main or called program as a web transaction.

---

## **EGL Java runtime error code VGJ0052E**

**VGJ0052E: User Interface record %1 contains too much data for it to be sent to the gateway servlet.**

### **Explanation**

The data of the record is too large to fit in the gateway servlet's buffer.

### **User Response**

Make the record smaller and regenerate the program.

---

## **EGL Java runtime error code VGJ0053E**

**VGJ0053E: An error occurred while communicating with the gateway servlet. Exception: %1 Message: %2.**

### **Explanation**

The exception was thrown during a CONVERSE or SHOW with a User Interface record.

### **User Response**

Use the exception and its message to diagnose and fix the problem.

---

## **EGL Java runtime error code VGJ0054E**

**VGJ0054E: The program was unable to verify data from the gateway servlet. The data identifier is %1.**

### **Explanation**

This is an internal error. The program expected one kind of data but received another. The data identifier is used for IBM diagnostic purposes only.

### **User Response**

Record the message number, the message text and the situation in which this message occurs. Contact IBM support center to report the problem.

---

## **EGL Java runtime error code VGJ0055E**

**VGJ0055E: An error occurred on a call to program %1. The error code was %2 (%3).**

### **Explanation**

The error occurred during a call to a local Java program.

### **User Response**

Use the exception message to diagnose and fix the problem.

---

## **EGL Java runtime error code VGJ0056E**

**VGJ0056E: Called program %1 expected %2 parameters but was passed %3.**

### **Explanation**

The wrong number of parameters was passed to a called program.

### **User Response**

Rewrite the calling program or the called program so that both expect the same number of parameters to be passed.

---

## **EGL Java runtime error code VGJ0057E**

**VGJ0057E: An exception occurred while passing parameters to called program %1. Exception: %2 Message: %3**

### Explanation

An error occurred during a call to a Java program. The error may have happened before or after the program began.

### User Response

Use the exception and its message to diagnose and fix the problem.

---

## EGL Java runtime error code VGJ0058E

**VGJ0058E: Properties file %1 could not be loaded.**

### Explanation

The program's properties file could not be loaded. The name of the properties file is obtained from the system property `vgj.properties.file`.

### User Response

Ensure that `vgj.properties.file` has the correct file name and that the properties file is in a Jar file or directory listed in the `CLASSPATH` environment variable.

---

## EGL Java runtime error code VGJ0060E

**VGJ0060E: StartTransaction to class %1 failed. The exception is %2.**

### Explanation

The exception was thrown while the program was attempting to start a new JVM to run the specified server class as a new transaction. The property `vgj.java.command` specifies the command used to start a new JVM. The default command is `java`.

### User Response

Ensure that the property `vgj.java.command` has the correct value, and that your program has permission to create a new process.

Put the `startTransaction` statement inside a try statement to prevent this from being a fatal error. When the `startTransaction` fails within a try statement, an error code will be stored in the `errorCode` system variable.

---

## EGL Java runtime error code VGJ0061E

**VGJ0061E: The web transaction was given input UI record %1, but it was defined with input UI record %2.**

### Explanation

The web transaction program was started with information from a user interface record bean that isn't known to the program.

## User Response

Ensure that the specified record is defined as the input UI record for the program. Regenerate the program and the Java Beans from the same user interface record definition.

---

## EGL Java runtime error code VGJ0062E

**VGJ0062E: One or more parameters passed to MQ program %1 was of the wrong type. %2**

### Explanation

An exception was thrown while attempting to call the MQ program. The parameters are incorrect.

### User Response

Consult the MQ program's documentation and the exception's message to correct the error.

---

## EGL Java runtime error code VGJ0064E

**VGJ0064E: Program %1 expected text form %2 but it was given text form %3 on a show statement.**

### Explanation

Both programs must use the same text form.

### User Response

Modify the programs to use the same text form and regenerate.

---

## EGL Java runtime error code VGJ0100E

**VGJ0100E: The data of %1 is not in %2 format.**

### Explanation

The data in the item is in an unexpected format. Another item may have written over the specified item.

### User Response

Correct the program logic to avoid the error.

---

## EGL Java runtime error code VGJ0104E

**VGJ0104E: %1 is not a valid index for subscript %2 of %3.**

### Explanation

One of the subscripts used with a multidimensional array is invalid. A subscript value must be between one and the number of occurrences defined for the subscripted item.

## User Response

Ensure that the index value is a valid subscript for the subscripted item.

---

## EGL Java runtime error code VGJ0105E

**VGJ0105E: %1 is not a valid index for %2.**

### Explanation

A subscript value must be between one and the number of occurrences defined for the subscripted item.

### User Response

Ensure that the index value is a valid subscript for the subscripted item.

---

## EGL Java runtime error code VGJ0106E

**VGJ0106E: User overflow during assignment of %1 to %2.**

### Explanation

The target of an assignment is not large enough to hold the result without truncating significant digits. The value of system variable VGVar.handleOverflow is 1, which causes the program to end.

### User Response

Do as follows:

- Increase the number of significant digits in the target; or
- Define the program logic to handle the overflow condition; use the system variables VGVar.handleOverflow and overflowIndicator.

---

## EGL Java runtime error code VGJ0108E

**VGJ0108E: HEX item %1 was assigned nonhexadecimal value %2.**

### Explanation

HEX items can receive only hexadecimal digits.

### User Response

Make sure that the source value includes only hexadecimal digits.

---

## EGL Java runtime error code VGJ0109E

**VGJ0109E: HEX item %1 was assigned nonhexadecimal value from %2: %3.**

### Explanation

HEX items can receive only hexadecimal digits.

## User Response

Make sure that the source in the assignment contains only hexadecimal digits.

---

### EGL Java runtime error code VGJ0110E

**VGJ0110E: HEX item %1 was compared to nonhexadecimal value: %2.**

#### Explanation

HEX items can be compared only to hexadecimal digits.

#### User Response

Make sure that the comparison value includes only hexadecimal digits.

---

### EGL Java runtime error code VGJ0111E

**VGJ0111E: HEX item %1 was compared to nonhexadecimal value from %2: %3.**

#### Explanation

HEX items can be compared only to hexadecimal digits.

#### User Response

Make sure that the comparison value contains only hexadecimal digits.

---

### EGL Java runtime error code VGJ0112E

**VGJ0112E: NUM item %1 was assigned nonnumeric value: %2.**

#### Explanation

NUM items can be assigned only numeric values. Such values contain digits and may have leading and trailing spaces, a decimal point, and a leading sign. The decimal point is allowed in between two digits, immediately before the first digit, or immediately after the last digit.

#### User Response

Make sure that the source value is numeric.

---

### EGL Java runtime error code VGJ0113E

**VGJ0113E: NUM item %1 was assigned nonnumeric value from %2: %3.**

#### Explanation

NUM items can be assigned only numeric values. Such values contain digits and may have leading and trailing spaces, a decimal point, and a leading sign. The decimal point is allowed in between two digits, immediately before the first digit, or immediately after the last digit.

### **User Response**

Make sure that the source value is numeric.

---

## **EGL Java runtime error code VGJ0114E**

**VGJ0114E: The value of item %1 (%2) is not valid as a subscript.**

### **Explanation**

The value has too many digits to be a subscript for any element in the array. A subscript value must be between one and the number of occurs declared for the structure item.

### **User Response**

Make sure that the index value is a valid subscript for the array.

---

## **EGL Java runtime error code VGJ0115E**

**VGJ0115E: %1 cannot be assigned a string. The string was %2.**

### **Explanation**

The item cannot be assigned a string.

### **User Response**

Do not assign a string to the item.

---

## **EGL Java runtime error code VGJ0116E**

**VGJ0116E: %1 cannot be assigned a number. The number was %2.**

### **Explanation**

The item cannot be assigned a number.

### **User Response**

Do not assign a number to the item.

---

## **EGL Java runtime error code VGJ0117E**

**VGJ0117E: %1 cannot be converted to a long.**

### **Explanation**

The item cannot be converted to a long.

### **User Response**

Complete the following steps:

1. Record the message number and the message text.



**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0118E

**VGJ0118E: %1 cannot be converted to a number.**

### Explanation

The item item cannot be converted to a number.

### User Response

Do not use the item in a place where a number is required.

---

## EGL Java runtime error code VGJ0119E

**VGJ0119E: %1 is not a valid number.**

### Explanation

While using the debugger, the user attempted to set the value of a numeric item, but the new value is not a number.

### User Response

Use a numeric value.

---

## EGL Java runtime error code VGJ0120E

**VGJ0120E: %1 is not a valid value for the starting index of the substring operator on item %2.**

### Explanation

The starting index cannot be less than 1 or more than the length of the item.

### User Response

Use a valid index for the starting index of the substring operator.

---

## EGL Java runtime error code VGJ0121E

**VGJ0121E: %1 is not a valid value for the ending index of the substring operator on item %2.**

### Explanation

The ending index cannot be less than 1 or more than the length of the item.

## User Response

Use a valid index for the ending index of the substring operator.

---

### EGL Java runtime error code VGJ0122E

**VGJ0122E: The ending index of the substring operator on item %1 is %2, which cannot be less than the starting index, which is %3.**

#### Explanation

The ending index of the substring operator cannot be less than the starting index.

#### User Response

Make sure that the starting index is less than or equal to the ending index.

---

### EGL Java runtime error code VGJ0123E

**VGJ0123E: The substring operator failed: %1 cannot be used as a string value.**

#### Explanation

The variable does not support the substring operator.

#### User Response

Change the program so that it does not use the substring operator on the variable.

---

### EGL Java runtime error code VGJ0124E

**VGJ0124E: %1 cannot be assigned a record. The record was %2.**

#### Explanation

The data item's type does not allow assignments from records.

#### User Response

Change the program so that it does not assign a record to the data item.

---

### EGL Java runtime error code VGJ0125E

**VGJ0125E: %1 cannot be used as a field.**

#### Explanation

The variable is not a field.

#### User Response

Change the program so that it does not use the variable as a field.

---

## EGL Java runtime error code VGJ0126E

**VGJ0126E: Incompatible types in comparison of %1 to %2.**

### Explanation

The types of the values are incompatible in a comparison.

### User Response

Ensure that the comparison uses compatible types.

---

## EGL Java runtime error code VGJ0127E

**VGJ0127E: %1 cannot be assigned a date or time value. The value was %2.**

### Explanation

The item cannot be assigned a date or time value.

### User Response

Do not assign a date or time value to the item.

---

## EGL Java runtime error code VGJ0140E

**VGJ0140E: Array function %1 failed because there was an attempt to expand array %2 beyond its maximum size.**

### Explanation

The array cannot hold any more values.

### User Response

Modify the program to check the size of the array before attempting to add to it.

---

## EGL Java runtime error code VGJ0141E

**VGJ0141E: %1 is an invalid index for array %2. Current size: %3. Max size: %4**

### Explanation

The index is out for range for the array.

### User Response

Modify the program to use a valid array index.

---

## EGL Java runtime error code VGJ0142E

**VGJ0142E: The maximumSize of array %1 cannot be changed. Expected %2 got %3.**

### **Explanation**

The array was passed on a call statement. The corresponding array in the called program had a different `maximumSize`.

### **User Response**

Change one of the programs so that both use an array with the same `maximumSize`.

---

## **EGL Java runtime error code VGJ0143E**

**VGJ0143E: %1 is not a valid size for array %2.**

### **Explanation**

The array was passed on a call statement. The called program changed the array's size to a value that is less than zero or larger than the value of the property `maxSize`.

### **User Response**

Change the programs so they use the same value for the property `maxSize`.

---

## **EGL Java runtime error code VGJ0144E**

**VGJ0144E: %1 failed for array %2. Too many sizes were specified.**

### **Explanation**

The specified function failed. Its argument is an array of sizes, which contained too many elements.

### **User Response**

Correct the argument.

---

## **EGL Java runtime error code VGJ0145E**

**VGJ0145E: %1 failed for array %2. The sizes must be numeric data items.**

### **Explanation**

The specified function failed. Its argument should be an array of numeric data items, but it was not.

### **User Response**

Correct the argument.

---

## **EGL Java runtime error code VGJ0146E**

**VGJ0146E: %1 failed for array %2. The size given was less than zero.**

### Explanation

The specified function failed. Its argument is an array of sizes. One of the sizes was less than zero, but this is not allowed.

### User Response

Correct the argument.

---

## EGL Java runtime error code VGJ0147E

**VGJ0147E: %1 failed for array %2. The maxSize given is less than the current size.**

### Explanation

The array's maxSize cannot be changed to a value that is less than its current size.

### User Response

Correct the argument.

---

## EGL Java runtime error code VGJ0160E

**VGJ0160E: Math function %1 failed with error code 8 (domain error).**

### Explanation

An argument to the function is not valid.

### User Response

Do as follows:

- Change the program logic to ensure that the arguments to the function are valid, as per the function's documentation; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0161E

**VGJ0161E: Math function %1 failed with error code 8 (domain error).**

### Explanation

The argument must be between -1 and 1.

### User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is between -1 and 1; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0162E

**VGJ0162E: Math function atan2 failed with error code 8 (domain error).**

### Explanation

Both arguments cannot be zero.

### User Response

Do as follows:

- Change the program logic to ensure that at least one argument passed to the function is not zero; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0163E

**VGJ0163E: Math function %1 failed with error code 8 (domain error).**

### Explanation

The second argument must not be zero.

### User Response

Do as follows:

- Change the program logic to ensure that the second argument is not zero; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0164E

**VGJ0164E: Math function %1 failed with error code 8 (domain error).**

### Explanation

The argument must be greater than zero.

### User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is greater than zero; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0165E

**VGJ0165E: Math function pow failed with error code 8 (domain error).**

## Explanation

If the first argument is zero, the second must be greater than zero.

## User Response

Do as follows:

- Change the program logic to ensure that if the first argument passed to the function is zero, the second argument is greater than zero; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0166E

**VGJ0166E: Math function pow failed with error code 8 (domain error).**

## Explanation

If the first argument is less than zero, the second must be an integer.

## User Response

Do as follows:

- Change the program logic to ensure that if the first argument passed to the function is less than zero, the second argument is an integer; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0167E

**VGJ0167E: Math function sqrt failed with error code 8 (domain error).**

## Explanation

The argument must be greater than or equal to zero.

## User Response

Do as follows:

- Change the program logic to ensure that the argument passed to the function is greater than or equal to zero; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0168E

**VGJ0168E: Math function %1 failed with error code 12 (range error).**

## Explanation

An intermediate or final result cannot be represented as a double precision floating point number or with the precision of the result item.

## User Response

Do as follows:

- Change the program logic to ensure that the target item is large enough to hold the result value; or
- Change the program logic so that the arguments to the function have values that do not cause this problem; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0200E

**VGJ0200E: String function %1 failed with error code 8.**

## Explanation

The index must be between 1 and the length of the string.

## User Response

Do as follows:

- Change the program logic to ensure that the index-related argument to the function ranges between 1 and the length of the string; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0201E

**VGJ0201E: String function %1 failed with error code 12.**

## Explanation

The length must be greater than zero.

## User Response

Do as follows:

- Change the program logic to ensure that the length arguments passed to the function have values that are greater than zero; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0202E

**VGJ0202E: String function `setNullTerminator` failed with error code 16.**



## Explanation

The last byte of the target string must be a blank or null character.

## User Response

Do as follows:

- Change the program logic to ensure that the last byte of the target string is a blank or null character; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0203E

**VGJ0203E: String function %1 failed with error code 20.**

## Explanation

The index of a DBCHAR or UNICODE substring must be odd so that the index identifies the first byte of a character.

## User Response

Do as follows:

- Change the program logic to ensure that the index arguments passed to the function are valid; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0204E

**VGJ0204E: String function %1 failed with error code 24.**

## Explanation

The length of a DBCHAR or UNICODE substring must be even to refer to a whole number of characters.

## User Response

Do as follows:

- Change the program logic to ensure that the length arguments passed to the function have valid values; or
- Call the function in a try statement, or set `VGVar.handleSysLibraryErrors` to 1 before calling the function, so that the program can handle the error.

---

## EGL Java runtime error code VGJ0215E

**VGJ0215E: %1 was passed the nonnumeric string %2.**

## Explanation

Every character in the portion of the string defined by the length argument must be numeric.

## User Response

Change the program logic so the characters in the portion of the string defined by the length argument are numeric.

---

## EGL Java runtime error code VGJ0216E

**VGJ0216E: %1 is not a valid date mask for %2.**

### Explanation

The date mask defined in the properties file for use with the function is not valid.

The valid characters for a date mask are as follows:

**D, M, Y**

D for Day, M for Month, Y for Year

### Separator character

Any nonnumeric, single-byte character except D, M, or Y.

Valid date masks can be in any of the following formats:

- Long Gregorian

The long version of the Gregorian mask must contain the following parts in any order:

**YYYY** 4-digit year

**MM** 2-digit numeric month

**DD** 2-digit numeric day of month

The mask parts must be separated by any nonnumeric single-byte character except D, M, or Y.

For example, a mask of YYYY/MM/DD is used to display August 25, 1997 as 1997/08/25.

- Long Julian

The long version of the Julian mask must contain the following parts in any order:

**YYYY** 4-digit year

**DDD** 3-digit numeric day of year

The mask parts must be separated by any single-byte nonnumeric character except D, M, or Y.

For example, a mask of DDD-YYYY can be used to display August 25, 1997 as 237-1997.

## User Response

Change the date mask property to a valid value and restart the program. If no date mask property is defined, a default date mask will be used.

The date masks can be set using the properties `vgj.datemask.gregorian.long.NNN` and `vgj.datemask.julian.long.NNN`, where *NNN* is the current NLS code.

---

## EGL Java runtime error code VGJ0217E

**VGJ0217E: An error occurred in the convert function with argument %1: %2**

### Explanation

The attempt to convert the data of the argument failed. The reason for the failure is included in the message.

### User Response

Use the error message to diagnose and correct the problem.

---

## EGL Java runtime error code VGJ0218E

**VGJ0218E: GetMessage failed. Could not find the message for key %1.**

### Explanation

No message was found for the key that was passed to the getMessage system function.

### User Response

Either add the message, or use a different key.

---

## EGL Java runtime error code VGJ0250E

**VGJ0250E: Could not retrieve item %1 from containing part %2.**

### Explanation

An internal error occurred. An attempt was made to access an item with the specified index in the record or table.

### User Response

Do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
- The type of internal error

2. Record the situation in which this message occurs.

3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0300E

**VGJ0300E: Table file for table %1 could not be loaded. Could not find a file named either %2 or %3.**

## Explanation

Neither of the named files could be found in any of the resource locations. All resource locations are searched for the first file. If no such file exists, all resource locations are searched for the second file.

Resource locations differ depending on the mechanism that was used to locate the table file.

If the error was encountered in an applet, resource locations refer to locations on the server machine and can vary depending on the implementation of the Java Virtual Machine. However, all implementations should search the directory on the server specified by the CODEBASE value. This value is set by the APPLET tag in the HTML file containing the applet. If no CODEBASE value is specified, it defaults to the directory on the web server containing the HTML file.

If the error was encountered in an application, valid resource locations are as follows:

- The directory that the Java Virtual Machine was started in (the working directory for the executable).
- Any directory specified in the CLASSPATH for the application being run. Specification of this value is system-dependent. On some systems, it can be specified as an environment variable. All systems allow it to be specified when invoking the Java Virtual Machine using the -classpath option. See the documentation that came with your copy of the Java Virtual Machine for more information on the value of CLASSPATH.

## User Response

First, locate the table file and make sure the permissions necessary to access it are set.

If the error occurred from within an applet or if the error occurred from within an application and you do not want to modify the existing set of resource locations, copy the table file into a valid resource location.

Otherwise, complete one of the following instructions:

- If the Java interpreter will use the value of the CLASSPATH environment variable, add the directory containing the table file to the current value of CLASSPATH.
- Specify the directory containing the table file by using the -classpath option when invoking the Java interpreter. If specifying the -classpath option overrides the value of the CLASSPATH environment variable, you need to specify the path to the Java runtime classes (e.g. classes.zip or rt.jar) in addition to any directories you add as resource locations.

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0301E

**VGJ0301E: Table file %1 for table %2 could not be loaded because an incorrect number of bytes was returned during the read operation on the table header.**

## Explanation

One of the following conditions exists:

- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

## User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0302E

**VGJ0302E: Table file %1 for table %2 could not be loaded because an unexpected magic number was encountered during inspection of the table header.**

## Explanation

One of the following conditions exists:

- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

## User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0303E

**VGJ0303E: Table file %1 for table %2 could not be loaded because an internal I/O error occurred during a read or close operation.**

### Explanation

One of the following conditions exists:

- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

### User Response

Regenerate the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0304E

**VGJ0304E: Table file %1 for table %2 could not be loaded because an incorrect number of bytes was returned during the read operation on the table data.**

### Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the program that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during runtime code generation.
- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

### User Response

Do as follows:

- If the column definition has not been changed, regenerate the table.
- If the column definition has been changed, either remove the change and regenerate the table or regenerate the runtime code for the program that uses the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0305E

**VGJ0305E: Table file %1 for table %2 could not be loaded. The data encountered in the table file for item %3 is not in the correct format. The corresponding data format error is: %4**

### Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the applet or application that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during runtime code generation.
- The table file has become corrupt.
- The table file was not generated with Rational Application Developer for z/OS or with VisualAge Generator.

### User Response

Do as follows:

- If the column definition has not been changed, regenerate the table.
- If the column definition has been changed, either remove the change and regenerate the table or regenerate the runtime code for the program that uses the table.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0306E

**VGJ0306E: Table file %1 for table %2 could not be loaded because the data in the table file is for a different type of table than table %2.**

### Explanation

One of the following conditions exists:

- The table file was regenerated after its columns were changed but the applet or application that is attempting to load the table was not regenerated. Generating only the table after changing the column definition causes an inconsistency to exist between the definition in the table file and the definition in the table class file, which is only generated during runtime code generation.
- The table file has become corrupt.
- The table file was not generated with EGL or VisualAge Generator.

### User Response

Do as follows:

- If the table type has not been changed, regenerate the table.
- If the table type has been changed, either edit the table definition so that it is of the correct type and regenerate the table or regenerate the runtime code for the program that uses the table.

Additional diagnostic information may become available if you enable program trace.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0307E

**VGJ0307E: Table file %1 for table %2 could not be loaded because table file %1 is a VisualAge Generator C++ table file and is not in big-endian format.**

### Explanation

Table files generated by the VisualAge Generator C++ generator can only be used with Java programs if the byte-ordering used to encode numeric data within the table is big-endian.

### User Response

Regenerate the table in big-endian format or as a Java platform-independent table.



To regenerate the table in big-endian format, use VisualAge Generator to generate the table for a C++ target system that is big-endian (e.g. AIX). To regenerate the table as a Java platform-independent table, generate the table for a Java target system with VisualAge Generator or EGL.

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0308E

**VGJ0308E: Table file %1 for table %2 could not be loaded. Table file %1 is a VisualAge Generator C++ table file, and the character encoding used in the table (%3) is not supported on the runtime system.**

### Explanation

Table files generated by the VisualAge Generator C++ generator can be used with Java programs only if the type of character encoding used for data within the table is the same type of encoding used by the runtime system.

### User Response

Do as follows:

1. Determine the character encoding used on your system. Java programs use either the ASCII or EBCDIC character encodings. Most workstations use the ASCII encoding. Most host platforms use the EBCDIC encoding. If you do not know the encoding used on your system, contact your system administrator.
2. Regenerate the table using the correct character encoding or as a Java platform-independent table.

To regenerate the table using the correct character encoding, use VisualAge Generator to generate the table for your target system or another C++ target system that uses the same character encoding. To regenerate the table as a Java platform-independent table, generate the table for a Java target system with VisualAge Generator or EGL.

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0315E

**VGJ0315E: A shared table entry for table %1 could not be found during the table unloading process.**

### Explanation

An internal error occurred.

### User Response

Do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred

- The type of internal error
- 2. Record the situation in which this message occurs.
- 3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0320E

**VGJ0320E: An edit routine with table %1 failed while comparing the table column %2 and the field %3.**

### Explanation

The table column and the field have types that are not valid for comparison.

### User Response

Do one of the following:

- Ensure that the types of the column and the field are valid for comparison by doing the following:
  1. Correct either the type of the column or the type of the field so that the comparison will be valid.
  2. Regenerate the program.
  3. Run the program.
- Modify your program to use a different table for the edit routine such that the comparison of the column and the field will be valid.

Refer to the trace output for more information.

---

## EGL Java runtime error code VGJ0330E

**VGJ0330E: Could not find a message with ID %1 in the message table %2.**

### Explanation

This error can occur during the following operations:

- Lookup of the the value for a form's msgField.
- Lookup of the value with the identifier specified as an edit message.

One of the following conditions exists:

- A message with this ID does not exist in the message table.
- The table file or message resource bundle for the table has become corrupt.

### User Response

Do one of the following:

- Ensure that a message with the message ID exists by doing the following:
  1. Add a message to the table with the message ID if it does not already exist.
  2. Regenerate the table.
  3. Run the program.
- Modify your program to use a different message that is already defined in the table.

- Modify your program to use a different message table that contains a message with the message ID.

---

## EGL Java runtime error code VGJ0331E

**VGJ0331E: Message table file %1 could not be loaded.**

### Explanation

The class for the program's message table could not be loaded, or an instance of the class could not be created.

### User Response

Ensure the message table has been generated.

---

## EGL Java runtime error code VGJ0350E

**VGJ0350E: An error occurred on a call to program %1. The error code was %2.**

### Explanation

A remote or EJB call to the specified program failed.

### User Response

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0351E

**VGJ0351E: commit failed: %1**

### Explanation

The resources could not be committed.

### User Response

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0352E

**VGJ0352E: rollBack failed: %1**

### Explanation

The resources could not be rolled back.

### User Response

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0400E

**VGJ0400E: An invalid parameter index, %1, was used for function %2.**

### Explanation

This is an internal error.

### User Response

Contact IBM support.

---

## EGL Java runtime error code VGJ0401E

**VGJ0401E: An invalid parameter descriptor was detected for function %1, parameter %2.**

### Explanation

This is an internal error.

### User Response

Contact IBM support.

---

## EGL Java runtime error code VGJ0402E

**VGJ0402E: The type of the value used for parameter %1 of function or program %2 is invalid.**

### Explanation

The value cannot be passed as a parameter, because the type of the value is incompatible with the type of the parameter.

### User Response

Do one of the following:

- Change the definition of the parameter to match the type of the value.
- Change the type of the value to match the definition of the parameter.

---

## EGL Java runtime error code VGJ0403E

**VGJ0403E: An error occurred while running script %1. The exception text is %2.**

### Explanation

The script caused an exception to be thrown.

### User Response

Correct the program logic to avoid the error.

---

## EGL Java runtime error code VGJ0416E

**VGJ0416E: An error occurred on a call to program %1. The error code was %2 (%3).**

### Explanation

An exception was thrown during an attempt to run the called program. The problem may be due to one of the following conditions:

- The program may not have permission to create a new process.
- The called program may not exist.
- The called program may not be found in the system path.

### User Response

Do as follows:

1. Verify that the program has permission to create a new process.
2. Verify that the called program exists.
3. Verify that the called program can be found in the system path.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ0450E

**VGJ0450E: I/O operation %1 with I/O object %2 failed for this reason: %3.**

### Explanation

An EGL I/O statement failed outside of a try statement, or when the value of system variable sysVar.handleHardIoErrors was zero.

### User Response

Review the error message and respond as appropriate.

---

## EGL Java runtime error code VGJ0500E

**VGJ0500E: No input received for required field - enter again.**

### Explanation

No data was typed in the field. The field is defined as required.

## **User Response**

Enter data in the field, or press a bypass edit key to bypass the edit check. Blanks will not satisfy the data input requirement for any type of field. In addition, zeros will not satisfy the data input requirement for numeric fields. The program continues.

---

## **EGL Java runtime error code VGJ0502E**

**VGJ0502E: Data type error in input - enter again.**

### **Explanation**

The data in the field is not valid numeric data. The field was defined as numeric.

### **User Response**

Enter only numeric data in this field, or press a bypass edit key to bypass the edit check. In either situation, the program continues.

---

## **EGL Java runtime error code VGJ0503E**

**VGJ0503E: Number of allowable significant digits exceeded - enter again.**

### **Explanation**

Data was entered into a numeric field that is defined with decimal places, a sign, currency symbol, or numeric separator edits. The input data exceeds the number of significant digits that can be displayed within the editing criteria. The number entered is too large. The number of significant digits cannot exceed the field length, minus the number of decimal places, minus the places required for editing characters.

### **User Response**

Enter a number with fewer significant digits.

---

## **EGL Java runtime error code VGJ0504E**

**VGJ0504E: Input not within defined range - enter again.**

### **Explanation**

The data in the field is not within the range of valid data defined for this item.

### **User Response**

Enter data that is within the defined range or press a bypass edit key to bypass the edit check. In either case, the program will continue.

---

## **EGL Java runtime error code VGJ0505E**

**VGJ0505E: Input minimum length error - enter again.**

### **Explanation**

The data in the field does not contain enough characters to meet the required minimum length.

### **User Response**

Enter the required number of characters to meet the minimum length or press a bypass edit key to bypass the edit check. In either case, the program will continue.

---

## **EGL Java runtime error code VGJ0506E**

**VGJ0506E: Table edit validity error - enter again.**

### **Explanation**

The data in the field does not meet the table edit requirement defined for the variable field.

### **User Response**

Enter data that conforms to the table edit requirement or press a bypass edit key to bypass the edit check. In either case, the program will continue.

---

## **EGL Java runtime error code VGJ0507E**

**VGJ0507E: Modulus check error on input - enter again.**

### **Explanation**

The data in the field does not meet the modulus check requirement defined for the variable field.

### **User Response**

Enter data that conforms to the modulus check defined for the variable field or press a bypass edit key to bypass the edit check. In either case, the program will continue.

---

## **EGL Java runtime error code VGJ0508E**

**VGJ0508E: Input not valid for defined date or time format %1.**

### **Explanation**

The data in the field, defined with a date edit, does not meet the requirements of the format specification.

### **User Response**

Enter the date in the correct format shown in the message.

---

## **EGL Java runtime error code VGJ0510E**

**VGJ0510E: Input not valid for boolean field.**

### Explanation

The value typed in the field does not conform to the boolean check. Input into a boolean field must be either 'Y' or 'N' for character fields and either 1 or 0 for numeric fields.

### User Response

Enter a 'Y' or 'N' for a character field or a 1 or 0 for a numeric field, or press the bypass edit key to bypass the edit check. In either case, the program will continue.

---

## EGL Java runtime error code VGJ0511E

**VGJ0511E: Edit table %1 is not defined for %2.**

### Explanation

A user message was requested but a user message table prefix was not defined for the program.

### User Response

Have the program developer do one of the following:

- Add the message table prefix to the program specification and generate the program again.
- Remove the user message number from the field edit and generate again.

---

## EGL Java runtime error code VGJ0512E

**VGJ0512E: Hexadecimal data is not valid.**

### Explanation

The data in the variable field must be in hexadecimal format. One or more of the characters you entered does not occur in the following set: a b c d e f A B C D E F 0 1 2 3 4 5 6 7 8 9

### User Response

Enter only hexadecimal characters in the variable field. The characters are left-justified and padded with the character 0. Embedded blanks are not permitted.

---

## EGL Java runtime error code VGJ0513E

**VGJ0513E: Value entered is invalid as it does not match the pattern that is set.**

### Explanation

Entered a value that does not match the pattern

### User Response

Enter the value as specified in the pattern.



---

## EGL Java runtime error code VGJ0514E

**VGJ0514E: Input maximum length error - enter again.**

### Explanation

Exceeded specified maximum length set for this field.

### User Response

Do not exceed the maximum length specified.

---

## EGL Java runtime error code VGJ0516E

**VGJ0516E: Input not within defined list - enter again.**

### Explanation

Input not within defined list - enter again.

### User Response

Input not within defined list - enter again.

---

## EGL Java runtime error code VGJ0517E

**VGJ0517E: Date/Time format specified %1 is invalid.**

### Explanation

The date or time format specified is invalid.

### User Response

Change the format to conform to the rules specified in the help topic *Date, time, and timestamp format specifiers*.

### Related reference

"Date, time, and timestamp format specifiers" on page 46

---

## EGL Java runtime error code VGJ0600E

**VGJ0600E: Unable to get linkage for program, %1.**

### Explanation

An entry for the specified program cannot be found in the CSO properties file because of one of the following reasons:

- An incorrect properties file was specified in the GatewayServlet configuration.
- The entry for the program was not specified in the CSO properties file.
- The CSO properties file is not in the directory specified in the GatewayServlet configuration.

## User Response

Contact the web server administrator to make sure that the following are performed:

- Make sure the GatewayServlet configuration specifies the correct CSO properties file using the linkageTable initialization parameter.
- Make sure that the program is defined in the CSO properties file.

---

## EGL Java runtime error code VGJ0601E

**VGJ0601E: An exception occurred while attempting to call entry point program, %1. Exception: %2. Message: %3.**

### Explanation

An unexplained error occurred while attempting to call the entry point program. The exception and message will define the error further. An entry point page or program gives the user a menu of programs which can be started using the GatewayServlet.

### User Response

Contact the web server administrator to make sure that the entry point page or the entry program are specified correctly in the GatewayServlet configuration.

---

## EGL Java runtime error code VGJ0603E

**VGJ0603E: The bean, %1, is invalid.**

### Explanation

The Page Bean or the bean name is invalid.

### User Response

Contact the web server administrator to make sure that the bean name is correct and that the Page Bean and the Java Server Page are deployed and made available to the GatewayServlet.

---

## EGL Java runtime error code VGJ0604E

**VGJ0604E: An exception occurred while attempting to load bean, %1. Exception: %2. Message: %3.**

### Explanation

An unexplained error occurred while trying to load the Page Bean. The exception and message will define the error further.

### User Response

Contact the web server administrator to make sure that the bean name is correct and that the Page Bean and the Java Server Page are deployed and made available to the GatewayServlet.

---

## EGL Java runtime error code VGJ0607E

**VGJ0607E: A version mismatch has occurred between the server, %1, and bean, %2.**

### Explanation

The version of the User Interface Record Bean does not match the version of the User Interface Record used by the server program. For proper operation, the versions must be compatible.

### User Response

Contact the program developer and generate both the program and user interface record beans. Contact the web server administrator to make sure that the user interface record bean is deployed to the proper location.

---

## EGL Java runtime error code VGJ0608E

**VGJ0608E: An error occurred while attempting to set data in the bean, %1. Exception: %2. Message: %3.**

### Explanation

An exception occurred while trying to set the record data from the server application into the User Interface Record Bean. The exception and message are included to help determine the problem.

### User Response

Use the exception and message included in the message for problem determination.

---

## EGL Java runtime error code VGJ0609I

**VGJ0609I: A gateway session is being bound for user, %1.**

### Explanation

This informational message appears on the application server's stdout or stderr. The message appears whenever a web session is created for the user.

### User Response

No response is required.

---

## EGL Java runtime error code VGJ0610I

**VGJ0610I: A gateway session is being unbound for user, %1.**

### Explanation

This informational message appears on the application server's stdout or stderr. The message appears whenever a web session has ended for the user. A session will end after a period of inactivity or if a severe error occurs that terminates the session.

## User Response

No response is required.

---

## EGL Java runtime error code VGJ0611E

**VGJ0611E: Unable to establish a connection with the SessionIDManager.**

### Explanation

The GatewayServlet was unable to connect to the SessionIDManager. The SessionIDManager is the component which gives session ids for gateway users. A session id is obtained for each active session and is used by the server program for saving and restoring application data.

The SessionIDManager is a separate application which listens for connects and requests for ids. When a session ends, the SessionIDManager will make the session id available to other sessions. The SessionIDManager must be active in order to run the GatewayServlet.

### User Response

Contact your web server administrator to start the SessionIDManager. If already started, the location of the SessionIDManager must be set in the GatewayServlet's configuration.

---

## EGL Java runtime error code VGJ0612I

**VGJ0612I: A gateway session is connected to the SessionIDManager for user, %1.**

### Explanation

This informational message appears in the web server's stdout or stderr. A session has connected to the SessionIDManager successfully in order to obtain a session id. The session id is used by the server program to save and restore program data.

### User Response

No response is required.

---

## EGL Java runtime error code VGJ0614E

**VGJ0614E: A required parameter, %1, is missing from the GatewayServlet configuration.**

### Explanation

A required parameter was not specified in the servlet configuration. The GatewayServlet will not run without these parameters.

### User Response

Contact the web server administrator to make sure that the GatewayServlet is properly configured. Reference your application server documentation to

determine how to configure servlet parameters.

---

## EGL Java runtime error code VGJ0615E

**VGJ0615E: Web transaction %1 is not allowed to run on this instance of the EGL Action Invoker.**

### Explanation

There was a problem creating or retrieving the GatewayRequestHandler for the program.

### User Response

Ensure that the named application has been generated and deployed to the server.

---

## EGL Java runtime error code VGJ0616E

**VGJ0616E: The gateway parameter %1 does not specify valid class: %2**

### Explanation

The class identified in the specified gateway property could not be loaded or instantiated.

### User Response

Ensure that the named class has been deployed to the server and specified correctly in the gateway properties file.

---

## EGL Java runtime error code VGJ0617E

**VGJ0617E: Please provide valid public user information in the gateway properties file.**

### Explanation

The public user name or password specified in the gateway properties file is invalid.

### User Response

Ensure that the public user name and password values in the gateway properties file are correct.

---

## EGL Java runtime error code VGJ0700E

**VGJ0700E: An error occurred during database connection: %1.**

### Explanation

An error occurred during an attempt to connect to a database. The error message ends with text from the database management system.

## User Response

Review the error message and respond as appropriate. Additional diagnostic information may become available if you enable program trace.

---

### EGL Java runtime error code VGJ0701E

**VGJ0701E: A database connection must be established prior to an SQL I/O operation.**

#### Explanation

An SQL I/O operation was attempted before a database connection was established.

#### User Response

An SQL I/O operation is valid only after the program creates a database connection. The program can create a default connection based on a program property and can override the default by running the connect system function. Review the EGL help pages for details on program properties and on setting up database access.

---

### EGL Java runtime error code VGJ0702E

**VGJ0702E: An error occurred during SQL I/O operation %1. %2.**

#### Explanation

An error occurred during the specified SQL I/O operation. The message ends with text from the database management system.

#### User Response

Review the message and respond as appropriate.

Additional diagnostic information may become available if you enable program trace.

---

### EGL Java runtime error code VGJ0703E

**VGJ0703E: An error occurred during setup for SQL I/O operation %1. %2.**

#### Explanation

An error occurred during setup for the specified SQL I/O operation.

#### User Response

Review the message and respond as appropriate.

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0705E

**VGJ0705E: An error occurred while disconnecting database %1. %2.**

### Explanation

An error occurred during an attempt to disconnect from the specified database. The error message ends with text from the database management system.

### User Response

Review the message and respond as appropriate.

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0706E

**VGJ0706E: Cannot set connection to database %1. The connection does not exist.**

### Explanation

An error occurred during an attempt to set the connection to the specified database. The connection can be set only to an active database connection within the transaction.

### User Response

Make sure that the name of the database matches one of the active database connections established for the transaction.

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0707E

**VGJ0707E: An SQL I/O sequence error occurred on %1.**

### Explanation

A sequence error may occur in these cases:

- An EGL replace or delete occurs but was not preceded by a setupd or update statement against the same SQL record
- An EGL get next occurs but was not preceded by a setupd or setinq statement against the same SQL record

The message identifies the last I/O operation that the program attempted, whether replace, delete, or scan.

### User Response

Make sure that the order of EGL statements is correct.

Additional diagnostic information may become available if you enable program trace.

---

## EGL Java runtime error code VGJ0708E

**VGJ0708E: Error while loading the JDBC driver classes: %1**

### Explanation

An error occurred while loading the JDBC driver classes, which are necessary for SQL I/O.

### User Response

Ensure that the JDBC driver classes are specified correctly in the property `vgj.jdbc.drivers`. If more than one is needed, separate their names with a semicolon. Also ensure that the classes can be found somewhere in the classpath.

---

## EGL Java runtime error code VGJ0709E

**VGJ0709E: A statement (%1) used a prepared statement that has not been prepared.**

### Explanation

The prepared statement named in the error message does not exist. Prepared statements are created by calling the EGL prepare statement.

### User Response

Correct the program logic by adding a prepare before the prepared statement is used.

---

## EGL Java runtime error code VGJ0710E

**VGJ0710E: A %1 statement used a result set that is closed or does not exist.**

### Explanation

The result set used by the statement cannot be used because it is not open or does not exist.

### User Response

Correct the program logic to avoid using invalid result sets.

---

## EGL Java runtime error code VGJ0711E

**VGJ0711E: An error occurred while connecting to database %1: %2**

### Explanation

A connection could not be established to the database named in the message.



## User Response

Use the Error part of this message to diagnose and correct the problem.

---

### EGL Java runtime error code VGJ0712E

**VGJ0712E: Cannot connect to the default database. The name of the default database was not specified.**

#### Explanation

The name of the default database was not specified, so the program cannot connect to it.

#### User Response

The name of the default database can be specified in several ways. One of the properties `vgj.jdbc.default.database.programName` (where `programName` is the name of the program) and `vgj.jdbc.default.database` must be set. The value of that property may be the actual name of the default database, or it may be the default database's logical name. When a logical name is used, another property must be set: `vgj.jdbc.database.logicalName`. The value of this property must be the actual name of the default database.

---

### EGL Java runtime error code VGJ0713E

**VGJ0713E: GET failed because result set %1 was not opened with scroll.**

#### Explanation

Only GET NEXT is allowed when the OPEN statement does not specify SCROLL.

#### User Response

Add the scroll option to the open statement where the result set is created.

---

### EGL Java runtime error code VGJ0750E

**VGJ0750E: The I/O driver for file %1 could not be created. %2**

#### Explanation

A failure occurred during creation of the I/O driver for the specified file. This error can occur at the following times:

- On the first I/O operation for a record that is related to the specified file; or
- On the first access of the system variable `resourceAssociation` for a record that is related to the specified file.

The end of the message indicates the reason for the failure.

#### User Response

Review the error message and respond as appropriate.

---

## EGL Java runtime error code VGJ0751E

**VGJ0751E: The fileType property for file %1 could not be found in the Java runtime property vgj.ra.fileName.fileType.**

### Explanation

You need to set the following runtime property to a valid file type:

`vgj.ra.fileName.fileType`

*fileName*

Name of the file specified in the message. This file name is a logical file name that is associated with an EGL record.

For an MQ record, the value is mq; for a serial record, the value is seqws. The source of the value is the generation-time resource associations part; specifically, the association element for the file, property **fileType**.

### User Response

Do as follows:

- Add the runtime fileType property to the runtime properties file or deployment descriptor; or
- Set the fileType value at generation time and regenerate the program:
  - In the file-name-specific association element of the resource associations part, set the property **fileType**
  - In the build descriptor used at generation, set the option **genProperties** to GLOBAL

For other details, see the EGL help pages on the association element, on Java runtime properties, and on setting up the environment.

---

## EGL Java runtime error code VGJ0752E

**VGJ0752E: An invalid fileType %1 was specified for file %2 in the resource associations part.**

### Explanation

You need to set the following runtime property to a valid file type:

`vgj.ra.fileName.fileType`

*fileName*

Name of the file specified in the message. This file name is a logical file name that is associated with an EGL record.

For an MQ record, the value is mq; for a serial record, the value is seqws. The source of the value is the generation-time resource associations part; specifically, the association element for the file, property **fileType**.

### User Response

Do as follows:

- Change the runtime fileType property in the runtime properties file or deployment descriptor; or
- Reset the fileType value at generation time and regenerate the program:

- In the file-name-specific association element of the resource associations part, change the property **fileType**
- In the build descriptor used at generation, set the option **genProperties** to GLOBAL

For other details, see the EGL help pages on the association element, on Java runtime properties, and on setting up the environment.

---

## EGL Java runtime error code VGJ0754E

**VGJ0754E: The record length item must contain a value that splits non-character data at item boundaries.**

### Explanation

The record has a variable length. When its data is written out, the record length item indicates how many bytes to write. The last byte of data must be the last byte of an item, unless the item is a char.

### User Response

Change the program so that the record length item's value points to the last byte of an item, or falls within a char item.

---

## EGL Java runtime error code VGJ0755E

**VGJ0755E: The value in the occursItem or lengthItem is too big.**

### Explanation

The record has a variable length. An attempt has been made to write out more bytes than the record currently contains.

### User Response

Change the program so that the value of the lengthItem or occursItem is within the size of the record.

---

## EGL Java runtime error code VGJ0770E

**VGJ0770E: An error occurred while creating the InitialContext or looking up the java:comp/env environment. The error was %1**

### Explanation

The exception was either thrown from the constructor of `javax.naming.InitialContext`, or from invoking the lookup method with the value `"java:comp/env"`. The program needs to create the `InitialContext` object and look up `"java:comp/env"` in order to access the J2EE environment settings.

### User Response

Use the text of the exception and the documentation of your J2EE environment to correct the problem.

---

## EGL Java runtime error code VGJ0800E

**VGJ0800E: The assignment of %1 to %2 is invalid.**

### Explanation

While using the debugger, you attempted to set a system variable to an invalid value.

### User Response

Choose a valid value, as described in the help page for the system variable.

---

## EGL Java runtime error code VGJ0801E

**VGJ0801E: %1 cannot be modified or does not exist.**

### Explanation

When using the debugger, you attempted to set the value of a system variable that cannot be set or does not exist.

### User Response

Review the help pages for a list of system variables and for a description of each.

---

## EGL Java runtime error code VGJ0802E

**VGJ0802E: Error debugging %1: %2**

### Explanation

An error occurred while attempting to debug a PageHandler or VGWebTransaction program.

### User Response

Use the Error part of the message to diagnose and correct the problem.

---

## EGL Java runtime error code VGJ0901E

**VGJ0901E: The date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.**

### Explanation

The date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.

### User Response

Adjust the date/time span pattern accordingly.

---

## EGL Java runtime error code VGJ0902E

**VGJ0902E: The precision of the date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.**

### Explanation

The precision date/time of the span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.

### User Response

Adjust the precision of the date/time span pattern accordingly.

---

## EGL Java runtime error code VGJ0903E

**VGJ0903E: The start code of the date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.**

### Explanation

The start code of the date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.

### User Response

Adjust the start code of the date/time span pattern accordingly.

---

## EGL Java runtime error code VGJ0904E

**VGJ0904E: The end code of the date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.**

### Explanation

The end code of the date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.

### User Response

Adjust the end code of the date/time span pattern accordingly.

---

## EGL Java runtime error code VGJ0905E

**VGJ0905E: Either the start code or the end code of the date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.**

### Explanation

Either the start code or the end code of the date/time span pattern (character string that declares the length and date/time components for a timestamp/interval item) is invalid.

### **User Response**

Adjust either the start code or the end code of the date/time span pattern accordingly.

---

## **EGL Java runtime error code VGJ0906E**

**VGJ0906E: The INTERVAL value is invalid.**

### **Explanation**

The INTERVAL value is invalid.

### **User Response**

Adjust the INTERVAL value accordingly.

---

## **EGL Java runtime error code VGJ0907E**

**VGJ0907E: The TIMESTAMP value is invalid.**

### **Explanation**

The TIMESTAMP value is invalid.

### **User Response**

Adjust the TIMESTAMP value accordingly.

---

## **EGL Java runtime error code VGJ0908E**

**VGJ0908E: The TIME value is invalid.**

### **Explanation**

The TIME value is invalid.

### **User Response**

Adjust the TIME value accordingly.

---

## **EGL Java runtime error code VGJ0909E**

**VGJ0909E: The DATE value is invalid.**

### **Explanation**

The DATE value is invalid.

### **User Response**

Adjust the DATE value accordingly.

---

## **EGL Java runtime error code VGJ0910E**

**VGJ0910E: The BLOB or CLOB is out of memory.**

**Explanation**

The BLOB or CLOB is out of memory.

**User Response**

Adjust the BLOB or CLOB size accordingly or associate it to file.

---

**EGL Java runtime error code VGJ0911E**

**VGJ0911E: An internal error occurred during the execution of loadTable. %1**

**Explanation**

An internal error occurred during the execution of loadTable.

**User Response**

For cause of error, see the extend error message.

---

**EGL Java runtime error code VGJ0912E**

**VGJ0912E: An SQL error occurred during the execution of loadTable. %1**

**Explanation**

An SQL error occurred during the execution of loadTable.

**User Response**

For cause of error, see the extend error message.

---

**EGL Java runtime error code VGJ0913E**

**VGJ0913E: An I/O error occurred during the execution of loadTable. %1**

**Explanation**

An I/O error occurred during the execution of loadTable.

**User Response**

For cause of error, see the extend error message.

---

**EGL Java runtime error code VGJ0914E**

**VGJ0914E: An error occurred during the loading of the VGJSystemCommandProcessing system library. %1**

**Explanation**

An error occurred during the loading of the VGJSystemCommandProcessing system library.

## **User Response**

For cause of error, see the extend error message.

---

### **EGL Java runtime error code VGJ0915E**

**VGJ0915E: System error occurred while executing the system command %1. Check your system's path whether the command exists, it is executable, etc.**

#### **Explanation**

An error occurred while executing the system command.

#### **User Response**

Check your system's path whether the command exists, it is executable, etc.

---

### **EGL Java runtime error code VGJ0916E**

**VGJ0916E: An internal error occurred during the execution of loadTable. %1**

#### **Explanation**

An internal error occurred during the execution of loadTable.

#### **User Response**

For cause of error, see the extended error message.

---

### **EGL Java runtime error code VGJ0917E**

**VGJ0917E: An SQL error occurred during the execution of unloadTable. %1**

#### **Explanation**

An SQL error occurred during the execution of unloadTable.

#### **User Response**

For cause of error, see the extended error message.

---

### **EGL Java runtime error code VGJ0918E**

**VGJ0918E: An I/O error occurred during the execution of unloadTable. %1**

#### **Explanation**

An I/O error occurred during the execution of unloadTable.

#### **User Response**

For cause of error, see the extend error message.



---

## EGL Java runtime error code VGJ0920E

**VGJ0920E: An error has been encountered while returning %1 from native C function.**

### Explanation

The error occurred while returning a value from C to EGL.

### User Response

This is an internal error.

---

## EGL Java runtime error code VGJ0921E

**VGJ0921E: An error has been encountered while passing %1 to native C function.**

### Explanation

The error occurred while passing a value from EGL to C.

### User Response

This is an internal error.

---

## EGL Java runtime error code VGJ0922E

**VGJ0922E: An error has been encountered while assigning value returned by native C function to %1.**

### Explanation

The error occurred while returning a value from C to EGL.

### User Response

This is an internal error.

---

## EGL Java runtime error code VGJ0923E

**VGJ0923E: Value too large to fit in %1.**

### Explanation

The number exceeds limits of smallint or int receiving variable.

### User Response

To store numbers that are outside the range of smallint or int, redefine the variable to use int or decimal type.

---

## EGL Java runtime error code VGJ0924E

**VGJ0924E: It is not possible to convert between the specified types.**

### **Explanation**

During native function calls, EGL attempts any data conversion that makes sense. Some conversions, however, are not supported, such as interval to date, timestamp to money etc.

### **User Response**

Check that you have specified the data types that you intended.

---

## **EGL Java runtime error code VGJ0925E**

### **VGJ0925E: The argument stack is empty.**

#### **Explanation**

An empty stack exception has been encountered while passing values to a native C function or while returning values from it.

#### **User Response**

Check that the number of receiving variables does not exceed the number of values passed or returned.

---

## **EGL Java runtime error code VGJ0926E**

### **VGJ0926E: Memory allocation failed.**

#### **Explanation**

Something in the current native function call required the allocation of memory, but the memory was not available.

#### **User Response**

Several things can cause this error. For example, your application is asking for more resources that the system is configured to allow, or a problem with the operating system requires that you reboot the system.

---

## **EGL Java runtime error code VGJ0927E**

### **VGJ0927E: Invalid datetime or interval qualifier.**

#### **Explanation**

An invalid qualifier has been used while receiving a timestamp or interval value in the native C function.

#### **User Response**

Check that you have specified the qualifier that you intended.

---

## **EGL Java runtime error code VGJ0928E**

### **VGJ0928E: Character host variable is too short for the data.**

### **Explanation**

A character host variable that is not large enough has been used while receiving a character string in the native C function.

### **User Response**

Check size of the variable.

---

## **EGL Java runtime error code VGJ0929E**

**VGJ0929E: The native C function, %1, was not found.**

### **Explanation**

The specified C function was not found in the function table.

### **User Response**

Add an entry for this function in the function table and recreate the shared library.

---

## **EGL Java runtime error code VGJ0930E**

**VGJ0930E: A loc\_t structure has been improperly modified in the native C code.**

### **Explanation**

A Clob or Blob data type has been passed to a native C function, but the loc\_t C structure in which it was received has been improperly changed.

### **User Response**

Check whether loc\_loctype, loc\_type or loc\_fname in the loc\_t structure have been changed in the native C code.

---

## **EGL Java runtime error code VGJ0931E**

**VGJ0931E: An error has occurred while processing a large object.**

### **Explanation**

The error occurred while performing some internal operation on a Clob or Blob data type.

### **User Response**

This is an internal error.

---

## **EGL Java runtime error code VGJ0932E**

**VGJ0932E: The native C function, %1, has not returned the correct number of values expected by the calling function.**

### **Explanation**

If the function was invoked as part of an expression, then it returned more than one value. Otherwise the number of returned variables was different from the number of receiving variables.

### **User Response**

Check that the correct function was called. Review the logic of the native C function, especially the values returned by it, to ensure that it always returns the expected number of values.

---

## **EGL Java runtime error code VGJ0933E**

**VGJ0933E: Intervals are incompatible for the operation.**

### **Explanation**

Some combinations of interval values are meaningless and are not allowed.

### **User Response**

Review the interval data types being passed or returned for compatibility.

---

## **EGL Java runtime error code VGJ0936E**

**VGJ0936E: Name of the native shared library has not been specified.**

### **Explanation**

The EGL application is calling a native C function, but name of the native shared library in which the C function resides has not been specified.

### **User Response**

Specify the native shared library name either using the `dllName` property of the EGL `nativeLibrary` part, or using the `vgj.defaultI4GLNativeLibrary` runtime property.

---

## **EGL Java runtime error code VGJ0937E**

**VGJ0937E: The native shared library %1 was not found or is not in the right format.**

### **Explanation**

The specified native shared library was not found in the library path or is not in the right format.

### **User Response**

Check if the library path has been set appropriately. The library path is a list of directories that the Java runtime system searches when loading shared libraries. Depending on runtime platform, set the library path using `PATH`, `LIBPATH`, `LD_LIBRARY_PATH` or `SHLIB_PATH` environment variable. If the library path is

appropriately set, check if the shared library has been created properly, or if a 64-bit shared library has been provided for a 32-bit application or vice versa.

---

## EGL Java runtime error code VGJ1000E

**VGJ1000E: %1 failed. Invoking a method or accessing a field called %2 resulted in an unhandled error. The error message is %3**

### Explanation

The error occurred in a Java access function. Either an Exception was thrown and the function was not called within a try statement or `VGVar.handleSysLibraryErrors` is 0, or something other than an Exception was thrown, such as an Error.

### User Response

Use information in the error message to correct the problem. If some kind of Exception was thrown, change the program logic to handle the error by calling the Java access function within a try statement, or by setting `VGVar.handleSysLibraryErrors` to 1 before invoking the Java access function.

---

## EGL Java runtime error code VGJ1001E

**VGJ1001E: %1 failed. %2 is not an identifier, or it is the identifier of a null object.**

### Explanation

The error occurred in a Java access function. The identifier cannot be used because it does not refer to a non-null object.

### User Response

Use an identifier of a non-null object.

---

## EGL Java runtime error code VGJ1002E

**VGJ1002E: %1 failed. A public method, field, or class named %2 does not exist or cannot be loaded, or the number or types of parameters are incorrect. The error message is %3**

### Explanation

The method, field, or class used by a Java access function could not be found.

### User Response

Do as follows:

- Make sure the target is a public method, field, or class.
- Make sure the name of the method, field, or class is correct. Class names must be qualified with the name of their package.
- If the problem is a missing class and the name is correct, make sure the directory or archive containing the class is in the Java classpath.

- If the problem is a missing method and the name is correct, make sure the types and number of parameters are correct. Compare the values passed to the Java access function with the values expected by the method.

---

## EGL Java runtime error code VGJ1003E

**VGJ1003E: %1 failed. The type of a value in EGL does not match the type expected in Java for %2. The error message is %3**

### Explanation

The type of a value passed to a Java access function is not correct.

### User Response

Values assigned to fields, and parameters passed to methods and constructors, must have the proper type. An exact match is not required as long as the conversion between the types is valid in Java. For example, a subclass may be used instead of its superclass, and a smaller primitive type, such as short, may be used instead of a larger one, such as int.

---

## EGL Java runtime error code VGJ1004E

**VGJ1004E: %1 failed. The target is a method that returned null, a method that does not return a value, or a field whose value is null.**

### Explanation

The Java access function expected the result of the operation to be a non-null object, but did not get one.

### User Response

To call a method that may return null or does not return a value, either use `javaStore`; or use the `java system` function and do not assign the result to an item. To get the value of a field that may be null, use `javaStoreField`.

---

## EGL Java runtime error code VGJ1005E

**VGJ1005E: %1 failed. The returned value does not match the type of the return item.**

### Explanation

The value returned by the Java access function cannot be assigned to the return item because of a type mismatch.

### User Response

Change the program logic to use a return item of an appropriate type.

---

## EGL Java runtime error code VGJ1006E

**VGJ1006E: %1 failed. The class %2 of an argument cast to null could not be loaded. The error message is %3**

## Explanation

The class of the argument passed to the Java access function could not be found.

## User Response

Do as follows:

- Make sure the name of the class is correct. Class names must be qualified with the name of a package.
- If the name is correct, make sure the directory or archive containing the class is in the Java classpath.

---

## EGL Java runtime error code VGJ1007E

**VGJ1007E: %1 failed. Could not get information about the method or field named %2, or an attempt was made to set the value of a field declared final. The error message is %3**

## Explanation

A `SecurityException` or `IllegalAccessException` was thrown while trying to get information about the method or field, or an attempt was made to set the value of a field declared final. Fields declared final cannot be modified.

## User Response

Do as follows:

- If the problem happened when setting a value, change the program logic so the code does not try to set the value of a field declared final; alternatively, change the declaration of the field.
- If the problem was access to information, ask a system administrator to update the security policy file of the Java Virtual Machine so that your program has the necessary permission. The administrator probably needs to grant the `ReflectPermission "suppressAccessChecks"`.

---

## EGL Java runtime error code VGJ1008E

**VGJ1008E: %1 failed. %2 is an interface or abstract class, so the constructor cannot be called.**

## Explanation

The constructor of an interface or abstract class cannot be called.

## User Response

Change the program logic to call the constructor of a class that is not abstract.

---

## EGL Java runtime error code VGJ1009E

**VGJ1009E: %1 failed. The method or field %2 is not static. An identifier must be used instead of a class name.**

### **Explanation**

When a method or field is not declared static, it exists only in a specific instance of a class, not the class itself. An identifier of the object must be used in this case.

### **User Response**

Change the program logic to use an identifier instead of a class name.

---

## **EGL Java runtime error code VGJ1101W**

**VGJ1101W: There are no more rows in the direction you are going.**

### **Explanation**

The end user tried to navigate beyond the last row.

---

## **EGL Java runtime error code VGJ1148E**

**VGJ1148E: Action field "%1" does not exist.**

### **Explanation**

The current OnEvent action refers to a field that cannot be found.

### **User Response**

Verify that the field exists in the current form.

---

## **EGL Java runtime error code VGJ1149E**

**VGJ1149E: Cannot insert another row - the input array is full.**

### **Explanation**

The variable used to hold the array data does not have space for another row.

### **User Response**

Increase the storage size of the EGL variable.

---

## **EGL Java runtime error code VGJ1150E**

**VGJ1150E: Array "%1" not found.**

### **Explanation**

The specified array could not be found in the ConsoleForm.

### **User Response**

Verify the array is correctly defined in the ConsoleForm and EGL program.



---

## EGL Java runtime error code VGJ1151E

**VGJ1151E: Assignment to prompt result variable failed.**

### Explanation

Assignment to prompt result variable failed.

### User Response

Verify the result variable can hold the result from the prompt action.

---

## EGL Java runtime error code VGJ1152E

**VGJ1152E: Screen Array Field "%1" size is incorrect.**

### Explanation

The specified screen array field size is not correct.

### User Response

Verify the definition of the screen array, and its usage in the EGL program.

---

## EGL Java runtime error code VGJ1153E

**VGJ1153E: DrawBox parameters are out of range.**

### Explanation

DrawBox parameters do not fit inside the current screen/window dimensions

### User Response

Verify the parameters to the drawbox function, and the current window dimensions.

---

## EGL Java runtime error code VGJ1154E

**VGJ1154E: Display coordinates are outside the window boundaries.**

### Explanation

Display coordinates are outside the window boundaries.

### User Response

Verify that the coordinates being used are within the size of the window.

---

## EGL Java runtime error code VGJ1155E

**VGJ1155E: Malformed key name "%1".**

### **Explanation**

The specified key name does not follow the key name convention.

### **User Response**

Rewrite the key name to follow the EGL key name conventions.

---

## **EGL Java runtime error code VGJ1156E**

**VGJ1156E: You cannot use this editing feature because a picture exists.**

### **Explanation**

The picture attribute restricts the editing features for this field.

### **User Response**

Use alternate editing keys and actions to obtain the desired results.

---

## **EGL Java runtime error code VGJ1157E**

**VGJ1157E: Cannot find window "%1".**

### **Explanation**

The window could not be located.

### **User Response**

Verify that the window is properly defined and used.

---

## **EGL Java runtime error code VGJ1158E**

**VGJ1158E: New window position/dimension values are invalid.**

### **Explanation**

The specified position/dimension values are not valid for the current display environment.

### **User Response**

Verify that the window position/dimensions are valid for the current display environment.

---

## **EGL Java runtime error code VGJ1159E**

**VGJ1159E: The command stack is out of sync.**

### **Explanation**

The statements being executed in the OnEvent clauses are causing EGL to become out of sync.

## **User Response**

Verify the usage of statements/function calls within the OnEvent block statements.

---

## **EGL Java runtime error code VGJ1160E**

**VGJ1160E: The Console UI library is not initialized.**

### **Explanation**

An attempt was made to use the Console UI library before it was initialized.

### **User Response**

Verify that the Console UI statement sequence is valid.

---

## **EGL Java runtime error code VGJ1161E**

**VGJ1161E: Illegal field type for construct.**

### **Explanation**

The field type specified in the console field is invalid for a construct query operation.

### **User Response**

Verify that the field type in the console field is valid for construct query operations..

---

## **EGL Java runtime error code VGJ1162E**

**VGJ1162E: ConstructQuery cannot be called with a variable list.**

### **Explanation**

A Construct Query operation was invoked with a variable list.

### **User Response**

Verify that the construct query operation is being invoked properly.

---

## **EGL Java runtime error code VGJ1163E**

**VGJ1163E: Cannot disable an invisible menu item.**

### **Explanation**

Attempt to hide an invisible menu item is an invalid operation.

### **User Response**

Verify that the correct menu item to be disabled is not an invisible menu item.

---

## **EGL Java runtime error code VGJ1164E**

**VGJ1164E: Edit action failed.**

### **Explanation**

The specified edit action failed to execute.

### **User Response**

Verify that the consolefield is properly defined, and the edit actions being performed are valid operations.

---

## **EGL Java runtime error code VGJ1165E**

**VGJ1165E: Error occurred while executing hotkey action.**

### **Explanation**

The hotkey operation failed to executed.

### **User Response**

Verify that the specified hotkey is valid, and the statement block also valid.

---

## **EGL Java runtime error code VGJ1166E**

**VGJ1166E: There is no active command to exit from.**

### **Explanation**

Attempt was make to exit the current command, which does not exist.

### **User Response**

Verify that the exit command is being used in the correct context.

---

## **EGL Java runtime error code VGJ1167E**

**VGJ1167E: There is no active command to continue.**

### **Explanation**

Attempt was made to continue the current command.

### **User Response**

Verify that the continue statement is being used in the correct context.

---

## **EGL Java runtime error code VGJ1168E**

**VGJ1168E: Fatal error: %1**

### **Explanation**

A Fatal runtime error occurred.

### **User Response**

Verify the Console UI statements are used in a proper context and sequence.

---

## **EGL Java runtime error code VGJ1169E**

**VGJ1169E: Field "%1" does not exist.**

### **Explanation**

The specified console field does not exist.

### **User Response**

Verify that the console field has been properly defined in the console form.

---

## **EGL Java runtime error code VGJ1170E**

**VGJ1170E: Screen array field "%1" is not an array.**

### **Explanation**

The referenced console field in the console form is not an array.

### **User Response**

Verify that the console field is defined as an array; verify that the correct console field is being referenced.

---

## **EGL Java runtime error code VGJ1171E**

**VGJ1171E: Field "%1" not found.**

### **Explanation**

The specified console field could not be found.

### **User Response**

Verify that the console field has been properly defined in the console form.

---

## **EGL Java runtime error code VGJ1172E**

**VGJ1172E: Cannot create ConsoleField without a window.**

### **Explanation**

An attempt was made to create a console field outside of a consoleform/window context.

### **User Response**

Verify the correctness of the consoleform and consolefield definitions.

---

## EGL Java runtime error code VGJ1173E

**VGJ1173E: Array field count mismatch.**

### Explanation

The Console UI array field specified does not match the referenced EGL array.

### User Response

Verify the ConsoleField and array definition; verify the correct EGL array variable is being used on the openui statement.

---

## EGL Java runtime error code VGJ1174E

**VGJ1174E: Form "%1" does not exist.**

### Explanation

The specified console form does not exist.

### User Response

Verify that the specified console form is defined and used in the correct context.

---

## EGL Java runtime error code VGJ1175E

**VGJ1175E: Form "%1" does not fit in window "%2".**

### Explanation

The form has dimensions that make it unable to fit into the current window dimensions.

### User Response

Alter the dimensions of either the form definition or the window definition.

---

## EGL Java runtime error code VGJ1176E

**VGJ1176E: Field lists do not match.**

### Explanation

The specified field list do not contain the same number of items as the variable list that was supplied.

### User Response

Alter the openUI statement to ensure that the same number of fields and variables are specified.

---

## EGL Java runtime error code VGJ1177E

**VGJ1177E: Form "%1" is busy.**

### **Explanation**

The form reference is currently already being used in another context.

### **User Response**

Verify that the EGL program logic uses a form only once at a time.

---

## **EGL Java runtime error code VGJ1178E**

**VGJ1178E: Form name "%1" already used.**

### **Explanation**

The definition of the form resulted in a form name conflict.

### **User Response**

Alter the Form definition to use a unique form name.

---

## **EGL Java runtime error code VGJ1179E**

**VGJ1179E: Form "%1" is not open.**

### **Explanation**

An attempt was made to reference a form object which is not defined.

### **User Response**

Verify that the specified form is properly defined, and used in valid ConsoleUI statements.

---

## **EGL Java runtime error code VGJ1180E**

**VGJ1180E: Cannot create ConsoleForm without a window.**

### **Explanation**

An attempt was made to create ConsoleForm without a valid window reference.

### **User Response**

Verify that the ConsoleForm is properly defined, and used within a valid ConsoleUI statement.

---

## **EGL Java runtime error code VGJ1181E**

**VGJ1181E: Cannot use KeyObject.getChar() for virtual keys.**

### **Explanation**

The consoleUI cannot use KeyObject.getChar() for virtual keys.

## **User Response**

Alter the EGL program to construct Strings for virtual key definitions.

---

### **EGL Java runtime error code VGJ1182E**

**VGJ1182E: Cannot use KeyObject.getCookedChar() for virtual keys.**

#### **Explanation**

The ConsoleUI cannot use KeyObject.getCookedChar() for virtual keys.

#### **User Response**

Alter the EGL program to use Strings to define virtual keys.

---

### **EGL Java runtime error code VGJ1183E**

**VGJ1183E: Retrieving prompt result string failed.**

#### **Explanation**

Retrieving prompt result string failed.

#### **User Response**

### **EGL Java runtime error code VGJ1184E**

**VGJ1184E: Help message key "%1" not found in resource bundle "%2".**

#### **Explanation**

The help message key could not be located within the specified message help file.

#### **User Response**

Verify that the correct help message key and help message file are being used.

---

### **EGL Java runtime error code VGJ1185E**

**VGJ1185E: Illegal array subscript.**

#### **Explanation**

An attempt was made to reference an invalid array element.

#### **User Response**

Verify that the program logic is referencing array elements within the size of the defined array.

---

### **EGL Java runtime error code VGJ1186E**

**VGJ1186E: Cannot initialize Console UI library.**



### **Explanation**

At program startup, the Console UI library could not be initialized.

### **User Response**

Verify that the program is being used in a supported display environment and platform.

---

## **EGL Java runtime error code VGJ1187E**

### **VGJ1187E: INTERNAL ERROR**

### **Explanation**

A ConsoleUI INTERNAL ERROR was encountered.

### **User Response**

---

## **EGL Java runtime error code VGJ1188E**

### **VGJ1188E: An INTERRUPT signal was received.**

### **Explanation**

An INTERRUPT signal was received.

### **User Response**

---

## **EGL Java runtime error code VGJ1189E**

### **VGJ1189E: Cannot have invisible menu item with no accelerator.**

### **Explanation**

An attempt was made to create an invisible menu item with no accelerator key.

### **User Response**

Alter the menu item definition to define an accelerator key for the invisible menu item.

---

## **EGL Java runtime error code VGJ1190E**

### **VGJ1190E: Cannot create a ConsoleLabel without a window.**

### **Explanation**

During the creation of the ConsoleLabel, a valid window reference could not be located.

### **User Response**

Verify that the console label is being correctly defined in the console form, and the console form correctly used in the EGL program.

---

## **EGL Java runtime error code VGJ1191E**

**VGJ1191E: Menu item %1 does not fit in window.**

### **Explanation**

The specified menu item is too large to fit into the current active window

### **User Response**

Alter the menu item so that the name is smaller than the current active window width dimension.

---

## **EGL Java runtime error code VGJ1192E**

**VGJ1192E: Menu item "%1" does not exist.**

### **Explanation**

The specified menu item could not be found or does not exist.

### **User Response**

Verify that the referenced menu item has been defined and added to the current menu instance.

---

## **EGL Java runtime error code VGJ1193E**

**VGJ1193E: Menu mnemonics conflict (key=%1).**

### **Explanation**

The current menu item definitions result in a mnemonic conflict.

### **User Response**

Alter the menu items to ensure that the accelerator/OnEvent keys do not conflict.

---

## **EGL Java runtime error code VGJ1194E**

**VGJ1194E: There is no active form.**

### **Explanation**

The console UI does not have an active form reference.

### **User Response**

Verify that a form has been defined and displayed.

---

## **EGL Java runtime error code VGJ1195E**

**VGJ1195E: Must have an active form for DISPLAY ARRAY.**

### **Explanation**

An attempt was made to display an array from the current active form which does not exist.

### **User Response**

Verify that a form has been defined with an array before attempting to display the array.

---

## **EGL Java runtime error code VGJ1196E**

**VGJ1196E: Must have an active form for READ ARRAY.**

### **Explanation**

An attempt was made to read an array from the active form, which does not exist.

### **User Response**

Verify that a form has been defined and made active before attempting to read an array from it.

---

## **EGL Java runtime error code VGJ1197E**

**VGJ1197E: Cannot start event loop with no current command.**

### **Explanation**

Cannot start event loop with no current command.

### **User Response**

---

## **EGL Java runtime error code VGJ1198E**

**VGJ1198E: No blob editor was specified.**

### **Explanation**

An attempt was made to edit a blob, but no blob editor was specified.

### **User Response**

Define an appropriate editor in the blob console field.

---

## **EGL Java runtime error code VGJ1199E**

**VGJ1199E: INTERNAL ERROR: No format object**

### **Explanation**

INTERNAL ERROR: No format object

## **User Response**

---

### **EGL Java runtime error code VGJ1200E**

**VGJ1200E: No Help File was specified.**

#### **Explanation**

A help request was received, but no help file was specified.

#### **User Response**

Define a valid help file in the EGL program.

---

### **EGL Java runtime error code VGJ1201E**

**VGJ1201E: No Help message was specified.**

#### **Explanation**

A help request was received, but no help message was specified.

#### **User Response**

Alter the EGL program to supply help messages.

---

### **EGL Java runtime error code VGJ1202E**

**VGJ1202E: Menu is not laid out.**

#### **Explanation**

An attempt was made to use Menu functions which has not been displayed.

#### **User Response**

Verify that the menu functions are used after the menu has been displayed.

---

### **EGL Java runtime error code VGJ1203E**

**VGJ1203E: There is no current screen array.**

#### **Explanation**

A reference was made to use the current screen array, which does not exist.

#### **User Response**

Verify that the current active form contains a screen array.

---

### **EGL Java runtime error code VGJ1204E**

**VGJ1204E: No menu items are visible.**

**Explanation**

During the construction of a menu, no menu items were found to be visible.

**User Response**

Alter the menu creation so that at least one menu item is visible and displayable.

---

**EGL Java runtime error code VGJ1205E**

**VGJ1205E: The name for the new window was null.**

**Explanation**

The declaration for the window was null.

**User Response**

Supply a window name when declaring a window.

---

**EGL Java runtime error code VGJ1206E**

**VGJ1206E: Attempt to open a Nil window.**

**Explanation**

An attempt was made to open a window and the window variable is Nil.

**User Response**

Verify the open window statement is using a valid window reference.

---

**EGL Java runtime error code VGJ1207E**

**VGJ1207E: An exception occurred in the prompt.**

**Explanation**

During the execution of a prompt, an exception occurred.

**User Response**

Verify the prompt OnEvent statement block is correct.

---

**EGL Java runtime error code VGJ1208E**

**VGJ1208E: A QUIT signal was received.**

**Explanation**

A QUIT signal was received.

## User Response

---

### EGL Java runtime error code VGJ1209E

**VGJ1209E: There is no active screen array.**

#### Explanation

The current active form does not contain a screen array.

#### User Response

Verify that the program logic is using a form that contains a screen array definition.

---

### EGL Java runtime error code VGJ1210E

**VGJ1210E: There is no active form.**

#### Explanation

The current Console UI session does not contain an active form instance.

#### User Response

Verify that a form is being defined and displayed before being referenced.

---

### EGL Java runtime error code VGJ1211E

**VGJ1211E: Menu cannot scroll to current item.**

#### Explanation

The attempt to move the menu cursor to a menu item failed.

#### User Response

Verify that the menu logic is correctly moving the menu cursor to the correct menu item. Verify that the menu item is not disabled.

---

### EGL Java runtime error code VGJ1212E

**VGJ1212E: Unknown attribute "%1"**

#### Explanation

The specified attribute was not recognized.

#### User Response

Verify that the attribute is correct for the current Console UI context.

---

### EGL Java runtime error code VGJ1213E

**VGJ1213E: Error in field "%1".**

### **Explanation**

The input into the field is incorrect.

### **User Response**

Verify that the typed in data matches the data type or format properties of the field.

---

## **EGL Java runtime error code VGJ1214E**

**VGJ1214E: Not enough variables were supplied.**

### **Explanation**

The openUI statement was not supplied with enough variables to bind to the console form.

### **User Response**

Alter the EGL program to list more variables for the openUI statement; alter the openUI statement to restrict the number of consolefields.

---

## **EGL Java runtime error code VGJ1215E**

**VGJ1215E: Window name "%1" is already used.**

### **Explanation**

The newly defined window name is already used by another window.

### **User Response**

Alter the name of the window to not conflict with other window names.

---

## **EGL Java runtime error code VGJ1216E**

**VGJ1216E: Window size is too small for help screen.**

### **Explanation**

An attempt to display the help screen into a display environment which is too small.

### **User Response**

Adjust the size of the display environment.

---

## **EGL Java runtime error code VGJ1217W**

**VGJ1217W: There are no more fields in the direction you are going.**

### **Explanation**

Attempt to move the cursor past the end of the field list.

## User Response

---

### EGL Java runtime error code VGJ1218E

**VGJ1218E: Screen array '%1' contents are invalid.**

#### Explanation

An on-screen arrayDictionary contains one entry for each column in the display. All entries must be the same type of object: either a ConsoleField or an array of ConsoleFields, and all arrays (if any) must have the same number of elements.

#### User Response

Examine and correct the declaration of the on-screen arrayDictionary.

---

### EGL Java runtime error code VGJ1219E

**VGJ1219E: Screen array '%1' cannot contain segmented field '%2'.**

#### Explanation

The consoleField **segment** property should not be set in any consoleField used in an on-screen arrayDictionary.

#### User Response

Remove the **segment** property in any consoleField included in an on-screen arrayDictionary.

---

### EGL Java runtime error code VGJ1220E

**VGJ1220E: Screen array '%1' is incompatible with the data array.**

#### Explanation

An on-screen arrayDictionary contains one entry for each column in the display, and a dynamic array is bound to that arrayDictionary. Each record in the arrayDictionary must have the same number of fields as the number of columns in the on-screen arrayDictionary, and each field must be assignment-compatible with the corresponding consoleField.

#### User Response

Examine and correct the on-screen arrayDictionary or the records in the dynamic array that is bound to that arrayDictionary.

---

### EGL Java runtime error code VGJ1221E

**VGJ1221E: Fields with same name '%1' detected.**

#### Explanation

A consoleForm cannot have more than one field with the same name.



### **User Response**

Correct the consoleForm declaration.

---

## **EGL Java runtime error code VGJ1222E**

**VGJ1222E: Console field '%1' length is invalid.**

### **Explanation**

The length of a consoleField must be greater than zero.

### **User Response**

Correct the consoleField declaration.

---

## **EGL Java runtime error code VGJ1223E**

**VGJ1223E: Label at [%1, %2] does not fit in the available space.**

### **Explanation**

A label must fit entirely within the boundaries of the window.

### **User Response**

Correct the position or size of either the label or window.

---

## **EGL Java runtime error code VGJ1224E**

**VGJ1224E: Field '%1' segment at (%2, %3) does not fit in the available space.**

### **Explanation**

A consoleField must fit entirely within the boundaries of the window.

### **User Response**

Correct the position or size of either the consoleField or the window.

---

## **EGL Java runtime error code VGJ1225E**

**VGJ1225E: Prompt string is too long for the active window.**

### **Explanation**

The message that prompts for user input must fit in the active window. If the `isChar` property of the prompt is set to `no`, you also must consider the space needed for the user's response.

### **User Response**

Correct the declaration of the prompt or window.

---

## EGL Java runtime error code VGJ1226E

**VGJ1226E: OpenUI array arguments were invalid.**

### Explanation

An on-screen arrayDictionary contains one entry for each column in the display. All entries must be the same type of object: either a ConsoleField or an array of ConsoleFields, and all arrays (if any) must have the same number of elements.

### User Response

Examine and correct the declaration of the on-screen arrayDictionary.

---

## EGL Java runtime error code VGJ1227E

**VGJ1227E: OpenUI field arguments were invalid.**

### Explanation

In the **openUI** statement, you can specify a list of consoleFields, either by specifying consoleFields or by specifying strings that contain the value of the consoleField name fields. In this case, the list included an invalid value.

### User Response

Examine and correct the values in the **openUI** statement.

---

## EGL Java runtime error code VGJ1228E

**VGJ1228E: Only a single variable may be bound to a prompt statement.**

### Explanation

A prompt must only be bound to a single variable that receives the user's response. Some other type of binding has been attempted.

### User Response

Examine and correct the **openUI** statement.

---

## EGL Java runtime error code VGJ1229E

**VGJ1229E: Could not determine data binding for console field '%1'.**

### Explanation

If an openUI statement does not bind consoleFields to other variables, the value of the **binding** property in each consoleField is used; but in this case, a **binding** property was not present.

### User Response

Add bindings in the consoleField or in the **openUI** statement.

---

## EGL Java runtime error code VGJ1290E

**VGJ1290E: %1 is not a valid parameter for the Blob/Clob function.**

### Explanation

An error occurred while processing a Blob/Clob function. The cause of the error is described in the message insert.

### User Response

Take appropriate action based on the content of the message insert.

---

## EGL Java runtime error code VGJ1301E

**VGJ1301E: Report Fill Error %1.**

### Explanation

Report Fill Error. The data provided to the report is not correct. The reasons could be that the dynamic array record field names do not match the report field names, the connection does not exist or the SQL statement is invalid.

### User Response

If you are using a dynamic array of records, please ensure that the field names defined in the report design match the elements in the record by name. If you are using a SQL statement, ensure that the SQL is valid. If you are using a connection, ensure that the connection has been established and the connection name is correct. In addition, make sure that the pathname specified for reportDesignFile is valid and that the file exists.

---

## EGL Java runtime error code VGJ1302E

**VGJ1302E: Report Export Error %1.**

### Explanation

Report Export Error. The report could not be exported to the specified format.

### User Response

Ensure that the pathnames are correct. the filled report object exists in the specified location and is correctly assigned in the reportDestinationFile field.

---

## EGL Java runtime error code VGJ1303E

**VGJ1303E: Report Dynamic Access Error, content not found. %1**

### Explanation

The field name does not exist in the dynamic array of records.

### **User Response**

Ensure that the field names match both in the report design and in the record that you are using in the EGL program.

---

## **EGL Java runtime error code VGJ1304E**

### **VGJ1304E: Incorrect connection name**

#### **Explanation**

The connection name is invalid.

#### **User Response**

Ensure that the connection is a valid EGL connection and the defineDatabaseAlias function has been used to assign a connection a name.

---

## **EGL Java runtime error code VGJ1305E**

### **VGJ1305E: Connection with specified name %1 does not exist**

#### **Explanation**

A connection with the connection name does not exist.

#### **User Response**

Ensure that the following statements are present in the EGL program. A connect function with valid parameters and a defineDatabaseAlias giving the connection a name.

---

## **EGL Java runtime error code VGJ1306E**

### **VGJ1306E: Incorrect EGL and Report type mapping. Check the mapping table.**

#### **Explanation**

There is a type mismatch between the fields in the Report Design and the data types in the EGL program.

#### **User Response**

Ensure that the types are compatible as mentioned in the documentation. Some examples, for a EGL char type, the design file should have the class defined as java.lang.String, for an EGL int type, the design file should have the field class as java.lang.Integer.

---

## **EGL Java runtime error code VGJ1401E**

### **VGJ1401E: Field "%1" at position(%2,%3) does not lie within the form.**

**Explanation**

The specified field does not lie within the form at the given position.

**User Response**

Verify that the form and fields are correctly defined.

---

**EGL Java runtime error code VGJ1402E**

**VGJ1402E: Field "%1" overlaps "%2".**

**Explanation**

The size and position of the two fields causes them to overlap.

**User Response**

Adjust the size and position coordinates of the fields.

---

**EGL Java runtime error code VGJ1403E**

**VGJ1403E: Internal error: Cannot determine form group.**

**Explanation**

Internal error: Cannot determine form group.

**User Response**

Verify that the form and formgroup are correctly defined.

---

**EGL Java runtime error code VGJ1404E**

**VGJ1404E: Form "%1" does not fit in any floating area.**

**Explanation**

The form does not fit in any floating area.

**User Response**

Verify that the form can be properly displayed in a floating area.

---

**EGL Java runtime error code VGJ1405E**

**VGJ1405E: Field "%1" coordinates are invalid.**

**Explanation**

The field coordinates are invalid.

**User Response**

Verify that the specified field coordinates are valid for the form.

---

## **EGL Java runtime error code VGJ1406E**

**VGJ1406E: Cannot get print association.**

### **Explanation**

The attempt to setup an print association failed.

### **User Response**

Verify that the printer association is setup correctly.

---

## **EGL Java runtime error code VGJ1407E**

**VGJ1407E: No suitable print device size exists.**

### **Explanation**

No suitable print device size exists.

### **User Response**

---

## **EGL Java runtime error code VGJ1408E**

**VGJ1408E: Printer '%1' was not found.\nThese printers are available:\n%2**

### **Explanation**

The user attempted to print to a specific printer device, which was not found in the system.

### **User Response**

Examine the printer configuration in the environment. Make sure the printer exists, or print to another printer.

---

## **EGL Java runtime error code VGJ1409E**

**VGJ1409E: No display device exists for forms.**

### **Explanation**

No display device exists for forms.

### **User Response**

Verify that the EGL program is being executed on a supported platform and display environment.

---

## **EGL Java runtime error code VGJ1410E**

**VGJ1410E: No compatible device size exists for displayed forms.**

**Explanation**

No compatible device size exists for displayed forms.

**User Response**

Verify that the EGL program is being executed on a supported platform and display environment.

---

**EGL Java runtime error code VGJ1411E**

**VGJ1411E: Help form class "%1" does not exist.**

**Explanation**

The attempt to reference the help form class, which does not exist.

**User Response**

Verify that the help form class is defined and referenced correctly.

---

**EGL Java runtime error code VGJ1412E**

**VGJ1412E: Unknown attribute "%1".**

**Explanation**

The specified attribute was not recognized.

**User Response**

Verify that the correct attribute name is being used.

---

**EGL Java runtime error code VGJ1414E**

**VGJ1414E: Cannot create help form '%1'**

**Explanation**

The help form could not be created for display.

**User Response**

Make sure the appropriate form groups and forms exist in the application and have been properly generated.

---

**EGL Java runtime error code VGJ1415E**

**VGJ1415E: INTERNAL ERROR: %1**

**Explanation**

An internal error has occurred.

## **User Response**

Contact IBM technical support.

---

### **EGL Java runtime error code VGJ1416E**

**VGJ1416E: There are no printers available.**

#### **Explanation**

The user attempted to print, but the system does not have printer devices to use.

#### **User Response**

Print to a file instead, or configure a printer device in the environment.

---

### **EGL Java runtime error code VGJ1417E**

**VGJ1417E: There is no default printer.**

#### **Explanation**

The user attempted to print to the default printer, but no default printer has been designated in the system.

#### **User Response**

Print to a specific printer, or define the default printer in the environment.

---

### **EGL Java runtime error code VGJ1419E**

**VGJ1419E: Formatted value '%1' for '%2' is longer than the maximum length (%3) allowed.**

#### **Explanation**

A formatted value (date, time, currency) is too long to fit in the field.

#### **User Response**

Increase the size of the field, or format the value to a shorter length.

---

### **EGL Java runtime error code VGJ1501E**

**VGJ1501E: Error loading property file %1**

#### **Explanation**

The error occurred while attempting to read the service properties file.

#### **User Response**

Verify the properties file is current, not in use by another application, and deployed to the correct location.



---

## EGL Java runtime error code VGJ1502E

**VGJ1502E: Error loading service properties for %1 from property file %2**

### Explanation

The error occurred while attempting to load the service properties from the specified file.

### User Response

Verify the properties file is current and deployed to the correct location.

---

## EGL Java runtime error code VGJ1503E

**VGJ1503E: Service binding error. The service is an EGL service and get/set Web Service properties is not valid, service:%1:property:%2.**

### Explanation

The error occurred while attempting to set/get a Web Service property on an EGL service. A setWebEndpoint or getWebEndpoint was issued against an EGL service. These method can only be used against a Web Service.

### User Response

Either change the service binding to a Web Service or change the get/set.

---

## EGL Java runtime error code VGJ1504E

**VGJ1504E: Service binding error. The service is a Web Service and get/set EGL service properties is not valid, service:%1:property:%2.**

### Explanation

The error occurred while attempting to set/get an EGL Service property on an Web Service. A setTCPIPLocation or getTCPIPLocation was issued against a Web Service. These methods can only be used against an EGL service.

### User Response

Either change the service binding to an EGL service or change the get/set.

---

## EGL Java runtime error code VGJ1525E

**VGJ1525E: Unable to find and load EGL service %1. Error message:%2**

### Explanation

The error occurred on the server during a local EGL service invocation. The service implementation could not be instantiated locally.

### **User Response**

Verify the service is specified correctly.

---

## **EGL Java runtime error code VGJ1526E**

**VGJ1526E: Unable to find and load remote EGL service %1**

### **Explanation**

The error occurred on the server during a remote EGL service invocation. The service implementation could not be instantiated on the server.

### **User Response**

Verify the service is specified correctly.

---

## **EGL Java runtime error code VGJ1527E**

**VGJ1527E: Unable to find and load WebService %1.**

### **Explanation**

The error occurred during a Web service invocation. The specified Web service could not be found or loaded.

### **User Response**

Verify the service is specified correctly and that the client has internet/intranet access.

---

## **EGL Java runtime error code VGJ1528E**

**VGJ1528E: Local EGL service error. Method with signature "%1 %2 (%3)" was not found on on service %4.**

### **Explanation**

The error occurred during a local EGL service invocation. The specified method could not be found on the service..

### **User Response**

Verify the service is specified correctly and both client and server are using the same interface.

---

## **EGL Java runtime error code VGJ1529E**

**VGJ1529E: Local EGL service error. Parameter error on method with signature "%1 %2 (%3)" on service %4 the parameters "(%5)" do not match the method signature. Error message:%6**

### **Explanation**

The error occurred during a local EGL service invocation. The method parameters on the invocation don't match the method parameters service.

## User Response

Verify the service is specified correctly and both client and server are using the same interface. Refer to the error message for other possible causes.

---

## EGL Java runtime error code VGJ1530E

**VGJ1530E: Local EGL service error. Method with signature "%1 %2 (%3)" on service %4 is inaccessible. Error message:%5**

### Explanation

The error occurred during a local EGL service invocation. The specified method is inaccessible.

### User Response

Refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1532E

**VGJ1532E: Remote EGL service error. The error occurred on the local end reading the return values. Remote EGL call to %1:%2, service:%3, method:"%4 %5 (%6)". Return code:%7. Error message:%8**

### Explanation

The error occurred on the client during a remote EGL service invocation. The client was unable to deserialize the return value.

### User Response

Check for network problems and refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1534E

**VGJ1534E: Remote EGL service error. The error occurred on the remote end while reading the header values. Remote EGL call to %1:%2, service:%3, method:"%4 %5 (%6)". Error message:%7**

### Explanation

The error occurred on the server during a remote EGL service invocation. The server encountered an error during de-serialization of the header message.

### User Response

Check for network problems and refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1535E

**VGJ1535E: Remote EGL service error. The error occurred on the remote end while writing the return values. Remote EGL call to %1:%2, service:%3, method:"%4 %5 (%6)". Error message:%7**

### Explanation

The error occurred on the server during a remote EGL service invocation. The server invoked the method but encountered an error during serialization of the return parameters.

### User Response

Check for network problems, verify that the client and server are using the same service interface and refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1536E

**VGJ1536E: Remote EGL service error. The error occurred on the remote end while initializing the parameter values. Remote EGL call to %1:%2, service:%3, method:"%4 %5 (%6)". Error message:%7**

### Explanation

The error occurred on the server during a remote EGL service invocation. The parameters were sent to the server but the server could not deserialize them.

### User Response

Check for network problems, verify that the client and server are using the same service interface and refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1537E

**VGJ1537E: An error occurred during the ACK handshake of the remote egl call to %1:%2, service:%3, method:"%4 %5 (%6)".**

### Explanation

The error occurred during a remote EGL service invocation. The parameters were received by the server and the server attempted to respond but the message was incomplete.

### User Response

Check for network problems and verify that the client and server are using the same service interface.

---

## EGL Java runtime error code VGJ1538E

**VGJ1538E: Local EGL service error. The error occurred on the local end while connecting or during data transfer. Remote EGL call to %1:%2, service:%3, method:"%4 %5 (%6)". Error message:%7**

### Explanation

The error occurred on the client during a remote EGL service invocation. The error occurred on the client end during the connect or the parameter serialization.

## User Response

Refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1539E

**VGJ1539E: Local EGL service error. The error occurred on the local end the reason can't be determined. Remote EGL call to %1:%2, service:%3, method:"%4 %5 (%6)". Error message:%7**

### Explanation

The error occurred on the client during a remote EGL service invocation. The error occurred on the client end but the cause could not be determined.

## User Response

Refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1540E

**VGJ1540E: Reference datatypes are not supported for remote services. Reference type parameter(s) %1. Remote call to service:%2, method:"%3 %4 (%5)".**

### Explanation

The error occurred during a remote EGL service invocation. The parameters specified in the error message are reference type parameters and reference type parameters are not supported in remote EGL services.

## User Response

Switch to the type from TCPIP to Local or redesign the service so it doesn't use reference data types.

---

## EGL Java runtime error code VGJ1541E

**VGJ1541E: Remote EGL service error. The error occurred on the remote end while initializing the parameter values. Remote EGL call to %1:%2, service:%3, method:"%4 %5 (%6)". The number of parameters sent was %7 the number of parameters in the service method are %8 (the return value is counted as a parameter).**

### Explanation

The error occurred on the server during a remote EGL service invocation. The number of method parameters is not the same as the number of method parameters on the server.

## User Response

Verify the method and service are correct. Verify the correct version of the service is deployed on the server. Verify that both client and server are using the same interface.

---

## EGL Java runtime error code VGJ1542E

**VGJ1542E: Remote EGL service error. Method:%1 was not found on on service %2. Error message:%3**

### Explanation

The error occurred on the server during a remote EGL service invocation. The specified method was not found on the specified service.

### User Response

Verify the method and service are correct. Verify the service is deployed on the server.

---

## EGL Java runtime error code VGJ1543E

**VGJ1543E: Unable to invoke WebService %1. \nError message:%2**

### Explanation

The call to the specified Web Service failed.

### User Response

Refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1544E

**VGJ1544E: Remote EGL service error. Method with signature "%1 %2 (%3)" on service %4 is inaccessible. Error message:%5**

### Explanation

The error occurred on the server during a remote EGL service invocation. The specified method is inaccessible.

### User Response

Refer to the error message for possible causes.

---

## EGL Java runtime error code VGJ1545E

**VGJ1545E: Remote EGL service error. Parameter error on method with signature "%1 %2 (%3)" on service %4 the parameters "(%5)" do not match the method signature.\nError message:%6**

### Explanation

The error occurred on the server during a remote EGL service invocation. The method parameters sent to the server don't match the method parameters on the server.

## User Response

Verify the service is specified correctly and both client and server are using the same interface. Refer to the error message for other possible causes.

---

## EGL Java runtime error code VGJ9900E

**VGJ9900E: An error has occurred. The error was %1. Unable to load the error description.**

### Explanation

The program either could not locate or load both the default message class file and the message class file for your locale. One or both of these message class files may be missing or corrupt.

**Note:** During run time, this message can only be displayed in U.S. English because of the problem in loading message files.

## User Response

If you have extracted class files from the file fda6.jar, verify that the classes you have are at the same release or maintenance level as the classes in the most recent file. If you are using older classes, replace them with the correct version. Also, you can reinstall fda6.jar from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
- The type of internal error

2. Record the situation in which this message occurs.

3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.

---

## EGL Java runtime error code VGJ9901E

**VGJ9901E: An error has occurred. The error was %1. The message text for %1 could not be found in the message class file %2. The message text for VGJ0002E also could not be found.**

### Explanation

The message class file does not contain the runtime message for the message ID or for message ID VGJ0002E. The message class file is either corrupt or from a previous release of EGL.

**Note:** During run time, this message can only be displayed in U.S. English because of the problem in loading message files.

## User Response

If you have extracted class files from the file `fda6.jar`, verify that the classes you have are at the same release or maintenance level as the classes in that file. If you are using older classes, replace them with the correct version. Also, you can reinstall `fda6.jar` from EGL.

If the problem persists, do as follows:

1. Record the message number and the message text.

**Note:** The error message includes the following important information:

- Where the error occurred
  - The type of internal error
2. Record the situation in which this message occurs.
  3. For further instructions on how to report possible defects to the IBM Support Center, refer to the *EGL Installation Guide*.





---

## Appendix. Notices

Note to U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106-0032, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:**

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
Lab Director  
IBM Canada Ltd. Laboratory  
8200 Warden Avenue  
Markham, Ontario, Canada L6G 1C7

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 2000, 2004. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

**Warning:** Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

---

## Trademarks and service marks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- AIX
- CICS
- CICS/ESA®
- ClearCase
- DB2
- IBM
- IMS
- Informix
- iSeries
- MQSeries
- MVS
- OS/400
- RACF
- Rational
- VisualAge
- WebSphere
- z/OS

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names, may be trademarks or service marks of others.

---

# Index

## Special characters

@DLI  
    fields 322  
@linkParms  
    primitive field-level property 798  
@PCB 145  
@programLinkData  
    primitive field-level property 797  
@Relationship 145  
@xsd  
    primitive field-level property 798

## A

abs() 967  
access parts  
    Interface 633  
acos() 968  
action  
    primitive field-level property 800  
activateWindow() 892  
activateWindowByName() 892  
activeForm 891  
activeWindow 892  
add statement  
    DL/I 662  
    SQL 664  
addReportData() 991  
addReportParameter() 991  
AIBTDLI() 930  
alias  
    primitive field-level property 801  
alias callLink element property 501  
alias names  
    Java 776  
    Java wrappers 776  
    overview 774  
alias transfer-related element  
    property 1089  
align  
    primitive field-level property 801  
ANY 37  
appendAll() 76  
appendElement() 76  
argument stack  
    C function 526, 528  
arrayDictionary part  
    description 87  
arrayIndex 1065  
arrays 75  
    dynamic arrays 75  
    functions 76  
        appendAll() 76  
        appendElement() 76  
        getMaxSize() 77  
        getSize() 77  
        insertElement() 77  
        removeAll() 77  
        removeElement() 77  
        resize() 78

arrays (*continued*)  
    functions (*continued*)  
        resizeAll() 78  
        setMaxSize() 78  
        setMaxSizes() 78  
    structure fields 79  
asin() 968  
Assignment compatibility 451  
assignments 456  
associations elements 457  
asynchLink elements  
    description 459  
    package 461  
    recordName 461  
atan() 969  
atan2() 969  
attachBlobToFile() 959  
attachBlobToTempFile() 960  
attachClobToFile() 960  
attachClobToTempFile() 960  
audit() 1018

## B

basic applications  
    starting 424  
basic program parts 842  
basic record parts 461  
BasicInterface 636  
batch interface for generation 417, 418  
beginDatabaseTransaction() 1020  
bidirectional language text  
    conversion 561  
BIGINT functions 519  
bindings 221  
BLOB 49  
breakpoints 379  
build descriptor parts  
    adding 387  
    description 383  
    editing general options 388  
    editing Java runtime properties 391  
    Java options 389  
    master build descriptors 386  
    options, alphabetic list 464  
    removing 392  
    setting the default 118  
build files  
    adding import statements 406  
    creating 383  
    description 15  
    editing import statements 406  
    format 462  
build parts  
    build descriptor 118, 383  
    linkage options 399  
    resource associations 393  
build paths, EGL  
    editing 407  
    overview 571

build plans  
    description 413  
    invoking after generation 419  
build project menu option 411  
build scripts  
    description 426  
    required options 498  
build servers  
    description 427  
    starting on AIX, Linux, or Windows  
        2000/NT/XP 427  
buildPlan build descriptor option 469  
byPassValidation  
    primitive field-level property 802  
bytes() 1020

## C

C data types 520  
C function  
    argument stack 526, 528  
    invoking 516, 520, 524  
    with EGL 516  
C functions  
    DATE 521  
    DATETIME 522  
    DECIMAL 523  
    INTERVAL 522  
calculateChkDigitMod10() 1021  
calculateChkDigitMod11() 1022  
call statement 665  
callCmd() 1023  
callConversionTable 1066  
callInterface 322  
callLink elements  
    alias 501  
    conversionTable 501  
    ctgKeyStore 502  
    ctgKeyStorePassword 502  
    ctgLocation 502  
    ctgPort 503  
    description 499  
    JavaWrapper 503  
    library 504  
    linkType 504  
    location 505  
    luwControl 506  
    package 507  
    parmForm 508  
    pgmName 509  
    providerURL 509  
    refreshScreen 510  
    remoteBind 510  
    remoteComType 511  
    remotePgmType 513  
    serverID 514  
    type 515  
cancelArrayDelete() 892  
cancelArrayInsert() 893  
capabilities, enabling 124  
case statement 668

- catcher program 356
- ceiling() 970
- CHAR 38
- characterAsInt() 999
- CICSJ2C call setup 441
- cicsj2cTimeout build descriptor
  - option 470
- clearActiveForm() 893
- clearActiveWindow() 893
- clearFields() 893
- clearFieldsByName() 894
- clearForm() 894
- clearRequestAttr() 931
- clearScreen() 917
- clearSessionAttr() 932
- clearWindow() 894
- clearWindowByName() 894
- clip() 999
- CLOB primitive type 48
- close statement 669
- closeActiveWindow() 895
- closeWindow() 895
- closeWindowByName() 895
- code generation, types 11
- code snippets
  - autoRedirect 180
  - databaseUpdate 182
  - getClickedRowValue 181
  - inserting 179
  - setCursorFocus 180
- color
  - primitive field-level property 802
- column
  - primitive field-level property 803
- command files 575
- commentLevel build descriptor
  - option 470
- commentLine 896
- comments 531
- comments, source code 363
- commit() 1024
- commitOnConverse 1057
- compareBytes() 1050
- compareNum() 970
- compareStr() 1000
- Complex properties 66
- concatenate() 1001
- concatenateBytes() 1052
- concatenateWithSeparator() 1002
- conditionAsInt() 1024
- connect() 1025
- ConnectionFactory, CICSJ2C 441
- connectionService() 1047
- Console UI
  - event handling 99
- Console UI variable
  - errorLine 903
- Console user interface overview 207
- ConsoleField
  - fields 533
  - properties 533
- ConsoleForm
  - part properties 546
- ConsoleLib
  - activateWindow() 892
  - activateWindowByName() 892
  - activeForm 891

- ConsoleLib (*continued*)
  - activeWindow 892
  - cancelArrayDelete() 892
  - cancelArrayInsert() 893
  - clearActiveForm() 893
  - clearActiveWindow 893
  - clearFields() 893
  - clearFieldsByName() 894
  - clearForm() 894
  - clearWindow() 894
  - clearWindowByName() 894
  - closeActiveWindow() 895
  - closeWindow() 895
  - closeWindowByName() 895
  - commentLine 896
  - currentArrayCount() 896
  - currentArrayDataLine() 896
  - currentArrayScreenLine() 897
  - currentDisplayAttrs 897
  - currentRowAttrs 897
  - cursorWrap 897
  - defaultDisplayAttributes 898
  - defaultInputAttributes 898
  - deferInterrupt 898
  - deferQuit 898
  - definedFieldOrder 899
  - displayAtLine() 899
  - displayAtPosition() 899
  - displayError() 900
  - displayFields() 900
  - displayFieldsByName() 900
  - displayForm() 901
  - displayFormByName() 901
  - displayLineMode() 901
  - displayMessage() 901
  - drawBox() 902
  - drawBoxWithColor() 902
  - errorLine 903
  - errorWindow 903
  - errorWindowVisible 903
  - formLine 903
  - getKey() 903
  - getKeyCode() 904
  - getKeyName() 904
  - gotoField() 904
  - gotoFieldByName() 905
  - gotoMenuItem() 905
  - gotoMenuItemByName() 905
  - hideAllMenuItems() 906
  - hideErrorWindow() 906
  - hideMenuItem() 906
  - hideMenuItemByName() 906
  - interruptRequested 907
  - isCurrentField() 907
  - isCurrentFieldByName() 907
  - isFieldModified() 907
  - isFieldModifiedByName() 908
  - key\_accept 908
  - key\_deleteLine 908
  - key\_help 909
  - key\_insertLine 909
  - key\_interrupt 909
  - key\_pageDown 909
  - key\_pageUp 909
  - key\_quit 909
  - lastKeyTyped() 910
  - menuLine 910

- ConsoleLib (*continued*)
  - messageLine 910
  - messageResource 910
  - nextField() 911
  - openWindow() 911
  - openWindowByName() 911
  - openWindowWithForm() 911
  - openWindowWithFormByName() 912
  - previousField() 912
  - promptLine 912
  - promptLineMode() 912
  - quitRequested 913
  - screen 913
  - scrollDownLines() 913
  - scrollDownPage() 913
  - scrollUpLines() 914
  - scrollUpPage() 914
  - setArrayLine() 914
  - setCurrentArrayCount() 914
  - showAllMenuItems() 915
  - showHelp() 915
  - showMenuItem() 915
  - showMenuItemByName() 915
  - sqlInterrupt 916
  - updateWindowAttributes() 916
- ConsoleUI
  - creating an interface 208
  - OpenUI statement 726
  - overview 209
- ConsoleUI overview 207
- ConsoleUI screen options
  - UNIX users 213
- constants, declarations 53
- constants, references to 59
- containsKey() 86
- content assist
  - description 577
  - using 131
- continue statement 672
- conversationID 1067
- converse statement 672
- ConverseLib
  - clearScreen() 917
  - displayMsgNum() 917
  - fieldInputLength() 918
  - pageEject() 918
  - validationFailed() 918
- ConverseVar
  - commitOnConverse 1057
  - eventKey 1058
  - printerAssociation 1059
  - segmentedMode 1061
  - validationMsgNum 1061
- conversion
  - bidirectional language text 561
  - data 558
- conversionTable callLink element
  - property 501
- convert() 1027
- copyBytes() 1053
- copyStr() 1003
- cos() 971
- cosh() 971
- Creating
  - subreports 263
- csouidpwd.properties 460



- ctgKeyStore callLink element
  - property 502
- ctgKeyStorePassword callLink element
  - property 502
- ctgLocation callLink element
  - property 502
- ctgPort callLink element property 503
- currency
  - primitive field-level property 804
- currencySymbol
  - primitive field-level property 805
- currencySymbol build descriptor
  - option 471
- currentArrayCount() 896
- currentArrayDataLine() 896
- currentArrayScreenLine() 897
- currentDate() 921
- currentDisplayAttrs 897
- currentFormattedGregorianCalendar 1078
- currentFormattedJulianDate 1079
- currentFormattedTime 1080
- currentGregorianCalendar 1080
- currentJulianDate 1081
- currentRowAttrs 897
- currentShortGregorianCalendar 1081
- currentShortJulianDate 1082
- currentTime() 921
- currentTimeStamp() 921
- curses library, UNIX 436
- cursors, SQL 277
- cursorWrap 897

## D

- data codes, SQL 874
- data conversion 558
- data initialization 564
- data parts
  - basic record 461
  - dataItem 133, 566
  - dataTable 176, 568
  - indexed record 632
  - MQ record 769
  - relative record 865
  - serial record 868
  - SQL record 877
- database authorization 557
- database connection preferences 121
- dataItem parts
  - creating 133
  - description 133
  - editing 134
  - EGL source format 566
- dataTable parts
  - creating 175
  - description 176
  - EGL source format 568
- DATE 41
- DATE functions 521
- Date, time, and timestamp format
  - specifiers 46
- dateFormat
  - primitive field-level property 805
- dateOf() 922
- datetime expressions 591
- DATETIME functions 522
- DateTimeLib 919

- DateTimeLib *(continued)*
  - currentDate() 921
  - currentTime() 921
  - currentTimeStamp() 921
  - dateOf() 922
  - dateValue() 922
  - dateValueFromGregorianCalendar() 923
  - dateValueFromJulian() 923
  - dayOf() 923
  - extend() 924
  - intervalValue() 924
  - intervalValueWithPattern() 925
  - mdy() 925
  - monthOf() 926
  - timeOf() 926
  - timestampFrom() 927
  - timestampValue() 927
  - timestampValueWithPattern() 928
  - timeValue() 928
  - weekdayOf() 929
  - yearOf() 929
- dateValue() 922
- dateValueFromGregorianCalendar() 923
- dateValueFromJulian() 923
- dayOf() 923
- DBCHAR 38
- dbms build descriptor option 471
- debugger, EGL
  - build descriptors 369
  - call statements 369
  - commands 369
  - creating a launch configuration 376
  - creating a Listener launch
    - configuration 376
  - invocation from generated code 369
  - overview 369
  - preparing a server 377
  - recommendations 369
  - setting preferences 116
  - SQL database access 369
  - starting a program 375
  - starting a server 377
  - starting a Web session 378
  - stepping through a program 380
  - system type preference 369
  - using breakpoints 379
  - viewing variables 380
- DECIMAL 50
- DECIMAL functions 523
- decimalSymbol build descriptor
  - option 471
- declaring
  - constants 53
  - redefined records 54
  - variables 53
- default database, SQL 298
- defaultDateFormat (system
  - variable) 1004
- defaultDateFormat build descriptor
  - option 472
- defaultDisplayAttributes 898
- defaultInputAttributes 898
- defaultMoneyFormat 1004
- defaultMoneyFormat build descriptor
  - option 472
- defaultNumericFormat 1005

- defaultNumericFormat build descriptor
  - option 472
- defaultTimeFormat 1005
- defaultTimestampFormat 1005
- defaultTimestampFormat build descriptor
  - option 473
- deferInterrupt 898
- deferQuit 898
- defineDatabaseAlias() 1029
- definedFieldOrder 899
- delete statement 673
  - DL/I 674
- deployment descriptors
  - setting values 438
  - updating 441
- deployment setup, J2EE
  - ConnectionFactory, CICSJ2C 441
  - descriptor values 438, 441
  - JDBC connections 445
  - runtime environment 437
  - TCP/IP listeners 436, 442
- deployment, Java applications outside of
  - J2EE 434
- Design document for reports
  - data types in 256
  - JasperReports 254
  - overview 251
- destDirectory build descriptor
  - option 473
- destHost build descriptor option 474
- destPassword build descriptor
  - option 475
- destPort build descriptor option 475
- destUserID build descriptor option 475
- development process 9
- dictionary
  - description 83
  - functions
    - containsKey() 86
    - getKeys() 86
    - getValues() 86
    - insertAll() 86
    - removeAll() 87
    - removeElement() 86
    - size() 87
  - properties 84
- DIF 351
- directories, generating into 420
- disconnect() 1030
- disconnectAll() 1030
- display statement 676
- displayAtLine() 899
- displayAtPosition() 899
- displayError() 900
- displayFields() 900
- displayFieldsByName() 900
- displayForm() 901
- displayFormByName() 901
- displayLineMode() 901
- displayMessage() 901
- displayMsgNum() 917
- displayName
  - primitive field-level property 807
- displayNames
  - primitive field-level property 808
- displayUse
  - primitive field-level property 808



- DL/I
  - add statement 662
  - Basic concepts 317, 325
  - delete statement 674
  - get next inParent statement 708
  - get next statement 702
  - get statement 690
  - replace statement 739
  - secondary index 327
  - specific tasks 326
- dliFieldName
  - primitive field-level property 809
- DLILib
  - AIBTDLI() 930
  - EGLTDLI() 930
  - psbData 931
- DOF 351
- drawBox() 902
- drawBoxWithColor() 902
- dynamic arrays 75
- dynamic SQL statements 288

**E**

- ear files, eliminating duplicate jar files 438
- editors
  - content assist 131, 577
  - EGL 577
  - locating source files 367
  - opening a part 367
  - preferences, EGL 118
- EGL build file format 462
- EGL build paths
  - editing 407
  - overview 571
- EGL command files 575
- EGL debugger
  - breakpoints 379
  - build descriptors 369
  - call statements 369
  - character encoding options 117
  - commands 369
  - creating a launch configuration 376
  - creating a Listener launch configuration 376
  - invocation from generated code 369
  - overview 369
  - preparing a server 377
  - recommendations 369
  - setting preferences 116
  - SQL database access 369
  - starting a program 375
  - starting a server 377
  - starting a Web session 378
  - stepping through a program 380
  - system type preference 369
  - viewing variables 380
- EGL editor
  - content assist 577
  - overview 577
  - preferences 118
- EGL form editor
  - display options 204
  - overview 194
  - preferences 204
- EGL Java runtime error codes 1097

- EGL overview 1
- EGL primitive types 520
- EGL properties
  - complex 66
  - overview 64
- EGL reserved words 581
- EGL runtime code for Java, installing 434
- EGL SDK (EGL Software Development Kit) 418
- EGL source format 586
- EGL\_GENERATORS\_PLUGINDIR variable 423
- EGLCMD 417, 572
- eglmaster.properties 585
- eglpsh 571
- EGLSDK 583
- EGLTDLI() 930
- EJB projects
  - deployment code generation 423
  - setting the JNDI name 441
- EJB sessions
  - components 402
  - description 403
- ELAISVN 356
- eliminateSystemDependentCode build descriptor option 476
- enableJavaWrapperGen build descriptor option 476
- environment files, J2EE
  - description 440
  - updating 440
- errorCode 1068
- errorLine 903
- errorLog() 1031
- errorWindow 903
- errorWindowVisible 903
- Event handling 99
- eventKey 1058
- exceptions
  - EGL system 94, 587
  - handling of 94
  - I/O error values 638
  - try blocks 94
- execute statement 677
- exit statement 681
- exp() 972
- explicit SQL statements 306, 307, 308
- exportReport() 992
- expressions
  - datetime 591
  - description 591
  - logical 88, 593
  - numeric 88, 600
  - string 88
  - text 601
- extend() 924
- externallyDefined transferToTransaction element property 1089

## F

- field-presentation properties 67
- fieldInputLength() 918
- fieldLen
  - primitive field-level property 809

- fields
  - ConsoleField 533
  - Menu 547
  - MenuItem 548
  - PresentationAttributes 550
  - properties 64
  - properties, page 792
  - properties, SQL 68
  - structure 880
  - Window 553
- Fields
  - Prompt 551
- file and database system words
  - recordName.resourceAssociation 985
- files
  - associations with record types 860
  - build 15, 383
  - creating 130, 383
  - deleting in the Project Explorer 368
  - EGL command 575
  - J2EE environment 440
  - linkage properties 447, 764
  - program properties 433
  - results 414
  - source 15, 130
  - Web service definition 15
- fill
  - primitive field-level property 810
- fillCharacter
  - primitive field-level property 810
- fillReport() 993
- findStr() 1006
- fixed record parts
  - description 136
- floatingAssign() 972
- floatingDifference() 973
- floatingMod() 973
- floatingProduct() 974
- floatingQuotient() 974
- floatingSum() 975
- floor() 975
- folders, creating 129
- for statement 683
- forEach statement 684
- form parts
  - creating a form in the EGL form editor 196
  - creating a print form 185
  - creating a text form 187
  - description 184
  - editing 195
  - EGL source format 606
  - field-presentation properties 67
  - filtering 196, 206
  - formatting properties 67
  - print 186
  - templates 200
  - text 188
  - validation properties 68
- formatDate() 1007
- formatNumber() 1007
- formatTime() 1008
- formatTimeStamp() 1009
- formatting properties 67
- formConversionTable 1069
- formGroup parts
  - creating 183

- formGroup parts *(continued)*
  - editing 195
  - EGL source format 603
  - pfKeyEquate property 792
  - use declarations 1094
- FormGroup parts
  - description 183
- formLine 903
- forward statement 686
- freeBlob() 961
- freeClob() 961
- freeSQL statement 687
- frexp() 976
- function invocations 613
- function parts 150, 621
  - creating 149
  - parameters 616
  - variables 615
- functions, Java access 935

## G

- genDataTables build descriptor
  - option 477
- genDirectory build descriptor
  - option 477
- generation
  - batch interface 417, 418
  - directory target 420
  - EGL command files 417
  - EGL SDK 418
  - EGLCMD 417, 572
  - eglpsh 571
  - EGLSDK 418, 583
  - EJB projects, deployment code 423
  - Java options 389
  - Java output 414, 781
  - Java wrappers 390
  - Java wrapper output 782
  - library parts 756
  - output types 625, 626
  - overview 409
  - Results view 627
  - setting
    - EGL\_GENERATORS\_PLUGINDIR 423
  - wizard 414
  - workbench 416
- genFormGroup build descriptor
  - option 478
- genHelpFormGroup build descriptor
  - option 478
- genProject build descriptor option 478
- genProperties build descriptor
  - option 480
- get absolute statement 695
- get current statement 697
- get first statement 698
- get last statement 699
- get next inParent
  - statement 707
- get next inParent statement
  - DL/I 708
- get next statement 701
  - DL/I 702
- get previous statement 708
- get relative statement 713
- get statement 687
- get statement *(continued)*
  - DL/I 690
- getBlobLen() 961
- getClobLen() 962
- getCmdLineArg() 1031
- getCmdLineArgCount() 1032
- getDataSource() 994
- getField() 942
- getFieldValue() 994
- getKey() 903
- getKeyCode() 904
- getKeyName() 904
- getKeys() 86
- getMaxSize() 77
- getMessage() 1032
- getNextToken() 1010
- getProperty() 1033
- getReportData() 994
- getReportParameter() 995
- getReportVariableValue() 995
- getRequestAttr() 932
- getSessionAttr() 933
- getSize() 77
- getStrFromClob() 962
- getSubStrFromClob() 962
- getTCPIPLocation() 987
  - ServiceLib 987
- getVAGSysType() 1054
- getValues() 86
- getWebEndPoint() 988
  - ServiceLib 988
- goTo statement 714
- gotoField() 904
- gotoFieldByName() 905
- gotoMenuItem() 905
- gotoMenuItemByName() 905
- Groups 264

## H

- handleHardIOErrors 1082
- handleOverflow 1083
- handler part 251
  - creating 266
- handleSysLibraryErrors 1084
- help
  - primitive field-level property 810
- HEX 38
- hideAllMenuItems() 906
- hideErrorWindow() 906
- hideMenuItem() 906
- hideMenuItemByName() 906
- hierarchy
  - PCB property 145
- highlight
  - primitive field-level property 811
- host variables, SQL 874

## I

- I/O error values 638
- I4GL data types 520
- if, else statement 715
- implicit SQL statements 305, 306, 307, 308
- import 33

- improving performance in EGL 10
- IMS Connect 356
- IMS-related considerations for EGL 627
- in EGL 1
- in operator 629
- indexed record parts 632
- Informix
  - special considerations 299
- initialization, data 564
- input forms 859
- input records 859
- inputRequired
  - primitive field-level property 811
- inputRequiredMsgKey
  - primitive field-level property 811
- insertAll() 86
- insertElement() 77
- installation, EGL runtime code for
  - Java 434
- integerAsChar() 1012
- intensity
  - primitive field-level property 812
- interface parts 633
- Interfaces
  - BasicInterface 636
  - JsonObject 155, 637
- interruptRequested 907
- INTERVAL 42
- INTERVAL functions 522
- intervalValue() 924
- intervalValueWithPattern() 925
- invoke() 944
- invoking
  - C function 520
- isa operator 641
- isBoolean
  - primitive field-level property 812
- isCurrentField() 907
- isCurrentFieldByName() 907
- isDecimalDigit
  - primitive field-level property 813
- isFieldModified() 907
- isFieldModifiedByName() 908
- isHexDigit
  - primitive field-level property 813
- isNull() 946
- isNullable
  - primitive field-level property 813
- isObjId() 947
- isReadOnly
  - primitive field-level property 814

## J

- J2EE build descriptor option 483
- J2EE deployment setup
  - ConnectionFactory, CICSJ2C 441
  - descriptor values 438, 441
  - JDBC connections 445
  - runtime environment 437
  - TCP/IP listeners 436, 442
- J2EE environment files
  - description 440
  - updating 440
- J2EE JDBC connections 445
- J2EELevel build descriptor option 484

- J2EELib
  - clearRequestAttr() 931
  - clearSessionAttr() 932
  - getRequestAttr() 932
  - getSessionAttr() 933
  - setRequestAttr() 933
  - setSessionAttr() 934
- jar files, runtime
  - eliminating duplicates from ear files 438
  - providing access to 448
- JasperReports 251, 254
- Java access functions 935
- Java alias names 776
- Java runtime properties 431, 642
- Java wrappers
  - alias names 776
  - classes 652
  - description 390
  - generating 390
  - generation output 782
  - using 11
- JavaLib
  - getField() 942
  - invoke() 944
  - isNull() 946
  - isObjId() 947
  - qualifiedTypeName() 948
  - remove() 949
  - removeAll() 950
  - setField() 951
  - store() 952
  - storeCopy() 954
  - storeField() 955
  - storeNew() 957
- JavaObject 155, 637
- JavaServer Faces
  - accessing components 232
  - component tree 229
  - controls and EGL 228
- JavaWrapper callLink element
  - property 503
- JDBC connections
  - J2EE 445
  - standard 309
- JDBC driver requirements in EGL 660
- JNDI name, setting for EJB projects 441
- JSPs 221

## K

- key\_accept 908
- key\_deleteLine 908
- key\_help 909
- key\_insertLine 909
- key\_interrupt 909
- key\_pageDown 909
- key\_pageUp 909
- key\_quit 909
- Keyboard shortcuts 131
- keyword statements
  - add 661
  - alphabetic list 91
  - alphabetical list 91
  - call 665
  - case 668
  - close 669

## keyword statements *(continued)*

- continue 672
- converse 672
- delete 673
- display 676
- execute 677
- exit 681
- for 683
- forEach 684
- forward 686
- freeSQL 687
- get 687
- get absolute 695
- get current 697
- get first 698
- get last 699
- get next 701
- get next inParent 707
- get previous 708
- get relative 713
- goTo 714
- if, else 715
- move 716
- MQSeries-related 339
- open 722
- prepare 736
- print 737
- replace 738
- return 741
- set 742
- show 751
- transfer 752
- try 754
- while 755

keywords

- new 212

## L

- lastKeyTyped() 910
- launch configurations
  - explicit 376
  - implicit 375
  - Listener 376
- Ldexp() 976
- library callLink element property 504
- library parts
  - creating 169
  - EGL source format 756
  - generated output 756
  - use declarations 1092
- library parts, type basicLibrary
  - description 169
- library parts, type nativeLibrary
  - description 171
- library parts, type ServiceBindingLibrary
  - description 172
- like 763
- limited-length string 40
- lineWrap
  - primitive field-level property 815
- linkage build descriptor option 484
- linkage options parts
  - adding 401
  - and IMS transaction codes 356
  - description 399
  - editing asynchLink elements 403

## linkage options parts *(continued)*

- editing callLink elements 401
- editing transfer-related elements 404
- removing 405
- linkage properties files
  - deploying 447
  - description 447
  - details 764
- linkType callLink element property 504
- loadBlobFromFile() 963
- loadClobFromFile() 963
- loadTable() 1033
- LobLib 958
  - attachBlobToFile() 959
  - attachBlobToTempFile() 960
  - attachClobToFile() 960
  - attachClobToTempFile() 960
  - freeBlob() 961
  - freeClob() 961
  - getBlobLen() 961
  - getClobLen() 962
  - getStrFromClob() 962
  - getSubStrFromClob() 962
  - loadBlobFromFile() 963
  - loadClobFromFile() 963
  - setClobFromString() 963
  - setClobFromStringAtPosition() 964
  - truncateBlob() 964
  - truncateClob() 965
  - updateBlobToFile() 965
  - updateClobToFile() 965
- location callLink element property 505
- log() 977
- log10() 977
- logic parts 869
  - basic program 842
  - function 150, 621
  - library 756
  - library, type basicLibrary 169
  - library, type nativeLibrary 171
  - library, type
    - ServiceBindingLibrary 172
  - PageHandler 223, 785
  - Service 869
  - textUI program 844
  - VGWebTransaction program 847
- logical expressions 593
- logical unit of work 395
- lowerCase
  - primitive field-level property 815
- lowerCase() 1012
- luwControl callLink element
  - property 506

## M

- masked
  - primitive field-level property 816
- master build descriptors
  - eglmaster.properties 585
  - overview 386
  - plugin.xml 602
- matches 766
- MathLib
  - abs() 967
  - acos() 968
  - asin() 968

MathLib (*continued*)  
 atan() 969  
 atan2() 969  
 ceiling() 970  
 compareNum() 970  
 cos() 971  
 cosh() 971  
 exp() 972  
 floatingAssign() 972  
 floatingDifference() 973  
 floatingMod() 973  
 floatingProduct() 974  
 floatingQuotient() 974  
 floatingSum() 975  
 floor() 975  
 frexp() 976  
 ldexp() 976  
 log() 977  
 log10() 977  
 maximum() 977  
 minimum() 978  
 modf() 978  
 pow() 979  
 precision() 979  
 round() 980  
 sin() 981  
 sinh() 981  
 sqrt() 982  
 stringAsDecimal() 982  
 stringAsFloat() 983  
 stringAsInt() 983  
 tan() 984  
 tanh() 984  
 maximum() 977  
 maximumSize() 1034  
 maxLen  
   primitive field-level property 816  
 MBCHAR 39  
 mdy() 925  
 Menu  
   fields 547  
 MenuItem  
   fields 548  
 menuLine 910  
 message customization for EGL Java  
   runtime 768  
 message queues 356  
   MQ options records 772  
   MQ record properties 772  
   MQSeries direct calls 341  
   MQSeries support 336  
   MQSeries-related EGL keywords 339  
   remote 340  
 messageLine 910  
 messageResource 910  
 MID 352  
 minimum() 978  
 minimumInput  
   primitive field-level property 816  
 minimumInputMsgKey  
   primitive field-level property 817  
 MOD 351  
 modf() 978  
 modified  
   primitive field-level property 817  
 modified data tags 191  
 monthOf() 926

move statement 716  
 MQ record parts  
   EGL source format 769  
   options records 772  
   properties 772  
 mqConditionCode 1084  
 MQSeries  
   direct calls 341  
   MQ options records 772  
   MQ record properties 772  
   related EGL keywords 339  
   support 336  
 msgTablePrefix build descriptor  
   option 484  
 multidimensional arrays 75

## N

names  
   aliases 774, 776  
   conventions 778  
 needsSOSI  
   primitive field-level property 818  
 newWindow  
   primitive field-level property 818  
 nextBuildDescriptor build descriptor  
   option 485  
 nextField() 911  
 non-fixed record parts  
   description 137  
 null 277  
 NUM 51  
 NUMC 52  
 numElementsItem  
   primitive field-level property 819  
 numeric expressions 600  
 numericFormat  
   primitive field-level property 819  
 numericSeparator  
   primitive field-level property 820

## O

one-dimensional arrays 75  
 open statement 722  
 OpenUI statement 726  
 openWindow() 911  
 openWindowByName() 911  
 openWindowWithForm() 911  
 openWindowWithFormByName() 912  
 operators  
   in 629  
   isa 641  
   precedence 779  
 options for generation  
   Java 389  
 outline  
   primitive field-level property 820  
 output  
   build project menu option 411  
   building 413  
   generated types 625, 626  
   Java generation 414, 781  
   Java wrapper generation 782  
   rebuild all menu option 411  
   rebuild project menu option 411

overflowIndicator 1069

## P

PACF 52  
 package asynchLink element  
   property 461  
 package callLink element property 507  
 packages  
   creating 130  
   description 15  
   recommendations for 15  
 Page Designer  
   bindings 221  
   check box components 246  
   command components 244  
   input components 245  
   multiple-selection components 248  
   output components 245  
   primitive types 242  
   Quick Edit view, page-handler  
     code 244  
   records 243  
   single-selection components 247  
   support 221  
 page field properties 792  
 pageEject() 918  
 pageHandler parts  
   binding check box components 246  
   binding command components 244  
   binding input components 245  
   binding multiple-selection  
     components 248  
   binding output components 245  
   binding single-selection  
     components 247  
   creating 221  
   EGL source format 785  
   use declarations 1095  
 PageHandler parts  
   description 223  
 parameters, function 616  
 parameters, program 840  
 parentRecord 145  
 parmForm callLink element  
   property 508  
 part properties  
   ConsoleForm 546  
 parts  
   description 19  
   filtering lists of 366  
   opening 367  
   properties 64  
   references to 23  
   searching for 363  
   viewing lists of 365  
 pattern  
   primitive field-level property 821  
 PCB  
   properties 145  
 pcbName 145  
 pcbType 145  
 performance tuning in EGL 10  
 persistent  
   primitive field-level property 821  
 pfKeyEquate property 792

- pgmName callLink element
  - property 509
- plugin.xml 602
- pow() 979
- precision() 979
- preferences
  - EGL 115
  - EGL debugger 116
  - EGL editor 118
  - EGL form editor 204
    - bidirectional text 205
    - palette entries 200
  - EGL-to-EGL migration 112
  - source styles 119
  - SQL database connections 121
  - SQL retrieve 123
  - templates 119
  - text 115
- prep build descriptor option 485
- prepare statement 736
- PresentationAttributes
  - fields 550
- previousField() 912
- primitive field-level properties 793
  - @linkParms 798
  - @programLinkData 797
  - @xsd 798
  - action 800
  - alias 801
  - align 801
  - byPassValidation 802
  - color 802
  - column 803
  - currency 804
  - currencySymbol 805
  - dateFormat 805
  - displayName 807
  - displayNames 808
  - displayUse 808
  - dliFieldName 809
  - fieldLen 809
  - fill 810
  - fillCharacter 810
  - help 810
  - highlight 811
  - inputRequired 811
  - inputRequiredMsgKey 811
  - intensity 812
  - isBoolean 812
  - isDecimalDigit 813
  - isHexDigit 813
  - isNullable 813
  - isReadOnly 814
  - lineWrap 815
  - lowerCase 815
  - masked 816
  - maxLen 816
  - minimumInput 816
  - minimumInputMsgKey 817
  - modified 817
  - needsSOSI 818
  - newWindow 818
  - numElementsItem 819
  - numericFormat 819
  - numericSeparator 820
  - outline 820
  - pattern 821
- primitive field-level properties (continued)
  - persistent 821
  - protect 822
  - runValidatorFromProgram 820
  - selectedIndexItem 823
  - selectFromListItem 823
  - selectType 824
  - sign 825
  - sqlDataCode 825
  - sqlVariableLen 826
  - timeFormat 827
  - timestampFormat 828
  - typeChkMsgKey 829
  - uiType 829
  - upperCase 831
  - validationOrder 831
  - validatorDataTable 832
  - validatorDataTableMsgKey 833
  - validatorFunction 833
  - validatorFunctionMsgKey 834
  - validValues 834
  - validValuesMsgKey 836
  - value 836
  - zeroFormat 837
- primitive types
  - ANY 37
  - BIN 50
  - BLOB 49
  - CHAR 38
  - CLOB 48
  - DATE 41
  - DBCHAR 38
  - DECIMAL 50
  - description 34
  - FLOAT 51
  - HEX 38
  - INTERVAL 42
  - MBCHAR 39
  - MONEY 51
  - NUM 51
  - NUMC 52
  - PACF 52
  - Page Designer 242
  - SMALLFLOAT 53
  - STRING 40
  - TIME 44
  - TIMESTAMP 44
  - UNICODE 41
- print forms 186
- print statement 737
- printerAssociation 1059
- program calls 11
- Program Communication Block (PCB) 317, 325
- program part
  - properties 856
- program parts
  - basic 842
  - creating 147
  - description 148
  - EGL source format 841
  - input forms 859
  - input records 859
  - Java generation 781
  - Java program generation 414
  - Java wrapper generation 782
  - non-parameter data 837
- program parts (continued)
  - parameters 840
  - textUI 844
  - use declarations 1092
  - VGWebTransaction 847
- program properties
  - @DLI 322
- program properties files 433
- Program Specification Block (PSB) 317, 325
- program transfers 11
- projects
  - creating 127
  - description 15
  - EJB, deployment code generation 423
  - EJB, JNDI name 441
  - specifying database options 129
- Prompt
  - Fields 551
- promptLine 912
- promptLineMode() 912
- properties
  - ConsoleField 533
  - field-presentation 67
  - fields 64
  - formatting 67
  - Java runtime 431, 642
  - MQ record 772
  - page field 792
  - parts 64
  - program part 856
  - SQL field 68
  - validation 68
  - variable-length records 860
- properties, primitive field-level
  - @linkParms 798
  - @programLinkData 797
  - @xsd 798
  - action 800
  - alias 801
  - align 801
  - byPassValidation 802
  - color 802
  - column 803
  - currency 804
  - currencySymbol 805
  - dateFormat 805
  - displayName 807
  - displayNames 808
  - displayUse 808
  - dliFieldName 809
  - fieldLen 809
  - fill 810
  - fillCharacter 810
  - help 810
  - highlight 811
  - inputRequired 811
  - inputRequiredMsgKey 811
  - intensity 812
  - isBoolean 812
  - isDecimalDigit 813
  - isHexDigit 813
  - isNullable 813
  - isReadOnly 814
  - lineWrap 815
  - lowerCase 815
  - masked 816



- properties, primitive field-level (continued)
  - maxLen 816
  - minimumInput 816
  - minimumInputMsgKey 817
  - modified 817
  - needsSOSI 818
  - newWindow 818
  - numElementsItem 819
  - numericFormat 819
  - numericSeparator 820
  - outline 820
  - pattern 821
  - persistent 821
  - protect 822
  - runValidatorFromProgram 820
  - selectedIndexItem 823
  - selectFromListItem 823
  - selectType 824
  - sign 825
  - sqlDataCode 825
  - sqlVariableLen 826
  - timeFormat 827
  - timestampFormat 828
  - typeChkMsgKey 829
  - uiType 829
  - upperCase 831
  - validationOrder 831
  - validatorDataTable 832
  - validatorDataTableMsgKey 833
  - validatorFunction 833
  - validatorFunctionMsgKey 834
  - validValues 834
  - validValuesMsgKey 836
  - value 836
  - zeroFormat 837
- protect
  - primitive field-level property 822
- providerURL callLink element
  - property 509
- PSB
  - scheduling 356
- psbData 931
- purge() 1035

## Q

- qualifiedTypeName() 948
- queryCurrentDatabase() 1035
- Quick Edit view
  - page-handler code 244
- quitRequested 913

## R

- rebuild all menu option 411
- rebuild project menu option 411
- receiving values from EGL 526
- recommendations, development
  - build descriptors 15
  - packages 15
  - part assignment 15
- record internals, SQL 876
- record parts
  - basic 461
  - creating 135

- record parts (continued)
  - description 135
  - indexed 632
  - MQ 769
  - Page Designer 243
  - properties, variable-length 860
  - relative 865
  - serial 868
  - SQL 277, 300, 301, 877
  - VGUI 1089
  - VGUIRecord 167
- record redefinition 54
- record types
  - associations with file types 860
  - description 138
- recordName asynchLink element
  - property 461
- redefines 54
- Reference compatibility 862
- reference types 212
- Reference types 863
- referencing
  - constants 59
  - parts 23
  - variables 59
- refreshScreen callLink element
  - property 510
- relative record parts 865
- remoteBind callLink element
  - property 510
- remoteComType callLink element
  - property 511
- remotePgmType callLink element
  - property 513
- remove() 949
- removeAll() 77, 87, 950
- removeElement() 77, 86
- replace statement 738
  - DL/I 739
- report handler 263
  - creating 266
- Report handler
  - functions 271
  - functions that you can invoke 272
  - overview 264
- Report library
  - overview 990
- Report part 262
- ReportData part 262
- ReportLib
  - addReportData() 991
  - addReportParameter() 991
  - exportReport() 992
  - fillReport() 993
  - getDataSource() 994
  - getFieldValue() 994
  - getReportData() 994
  - getReportParameter() 995
  - getReportVariableValue() 995
  - resetReportParameters() 996
  - setReportVariableValue() 996
- reports
  - code examples for driver
    - functions 260
  - code for invoking reports 257
  - creating 252
  - data source sample code 260

- reports (continued)
  - data types in XML design
    - documents 256
  - event handling 99
  - exported file formats 274
  - exporting 274
  - generating files to create 273
  - library 990
  - overview 251
  - overview of creating 252
  - report handler 264
  - templates for 259
  - writing report-driver code 257
  - XML design document 251, 254
- reserved words
  - EGL 581
- resetReportParameters() 996
- resize() 78
- reSizeAll() 78
- resource associations parts
  - adding 397
  - associations elements 457
  - description 393
  - editing 397
  - removing 398
- resourceAssociations build descriptor
  - option 486
- result-set processing, SQL 277, 867
- results files 414
- Results view, generation 627
- resultSetID 867
- retrieve feature, SQL 277, 299
- retrieve preferences, SQL 123
- return statement 741
- returnCode 1070
- returning values to EGL 528
- rollback() 1036
- round() 980
- run units 866
- runtime environment, J2EE setup 437
- runValidatorFromProgram
  - primitive field-level property 820

## S

- screen 913
- scrollDownLines() 913
- scrollDownPage() 913
- scrollUpLines() 914
- scrollUpPage() 914
- Secondary index
  - in DL/I 327
- secondaryTargetBuildDescriptor build
  - descriptor option 486
- segmentation
  - text applications 189
- segmentedMode 1061
- segmentRecord 145
- Segments
  - in DL/I 317, 325
- selectedIndexItem
  - primitive field-level property 823
- selectFromListItem
  - primitive field-level property 823
- selectType
  - primitive field-level property 824
- serial record parts 868

serverID callLink element property 514  
sessionBeanID build descriptor  
  option 488  
sessionID 1071  
set statement 742  
Set-value blocks 68  
setArrayLine() 914  
setBlankTerminator() 1012  
setClobFromString() 963  
setClobFromStringAtPosition() 964  
setCurrentArrayCount() 914  
setCurrentDatabase() 1036  
setError() 1037  
setField() 951  
setLocale() 1038  
setMaxSize() 78  
setMaxSizes() 78  
setNullTerminator() 1013  
setRemoteUser() 1039  
setReportVariableValue() 996  
setRequestAttr() 933  
setSessionAttr() 934  
setSubStr() 1013  
setTCPIPLocation() 988  
  ServiceLib 988  
setWebEndPoint() 989  
  ServiceLib 989  
show statement 751  
showAllMenuItems() 915  
showHelp() 915  
showMenuItem() 915  
showMenuItemByName() 915  
sign  
  primitive field-level property 825  
sin() 981  
sinh() 981  
size() 87, 1040  
snippets  
  autoRedirect 180  
  databaseUpdate 182  
  getClickedRowValue 181  
  inserting 179  
  setCursorFocus 180  
Software Development Kit, EGL (EGL SDK) 418  
source files  
  commenting 363  
  content assist 131, 577  
  creating 130  
  description 15  
  editors  
  commenting source code 363  
  format 586  
  locating in the Project Explorer 367  
  source assist 577  
source styles, preferences 119  
spaces() 1014  
SQL  
  constructing a PREPARE  
  statement 306  
  creating dataItem parts 300, 301  
  cursors 277  
  data codes 874  
  database authorization 557  
  database connection preferences 121  
  default database 298  
  dynamic statements 288  
  SQL (continued)  
  EGL statements 277  
  examples 288  
  explicit statements 277, 306, 307, 308  
  host variables 874  
  implicit statements 277, 305, 306, 307, 308  
  null 277  
  record internals 876  
  record parts 277  
  result-set processing 277, 867  
  retrieve feature 277, 299  
  retrieve preferences 123  
  support 277, 299  
  SQL field properties 68  
  SQL record parts 877  
  sqlca 1071  
  sqlcode 1072  
  sqlDataCode  
  primitive field-level property 825  
  sqlDB build descriptor option 490  
  sqlerrd 1085  
  sqlerrmc 1086  
  sqlID build descriptor option 491  
  sqlInterrupt 916  
  sqlIsolationLevel 1086  
  sqlJDBCDriverClass build descriptor  
  option 491  
  sqlJNDIName build descriptor  
  option 492  
  sqlPassword build descriptor option 492  
  sqlState 1073  
  sqlValidationConnectionURL build  
  descriptor option 493  
  sqlVariableLen  
  primitive field-level property 826  
  sqlWarn 1087  
  sqrt() 982  
  standard JDBC connections 309  
  startCmd() 1040  
  startLog() 1041  
  startTransaction() 1041  
  statements  
  assignment 88, 456  
  constant declaration 88  
  function invocation 88, 613  
  keyword 88  
  null 88  
  SQL 277  
  variable declaration 88  
  static arrays 75  
  store() 952  
  storeCopy() 954  
  storeField() 955  
  storeNew() 957  
  STRING 40  
  limited length 40  
  stringAsDecimal() 982  
  stringAsFloat() 983  
  stringAsInt() 983  
  strLen() 1014  
  StrLib  
  characterAsInt() 999  
  clip() 999  
  compareStr() 1000  
  concatenate() 1001  
  concatenateWithSeparator() 1002  
  StrLib (continued)  
  copyStr() 1003  
  defaultDateFormat 1004  
  defaultMoneyFormat 1004  
  defaultNumericFormat 1005  
  defaultTimeFormat 1005  
  defaultTimestampFormat 1005  
  findStr() 1006  
  formatDate() 1007  
  formatNumber() 1007  
  formatTime() 1008  
  formatTimeStamp() 1009  
  getNextToken() 1010  
  integerAsChar() 1012  
  lowerCase() 1012  
  setBlankTerminator() 1012  
  setNullTerminator() 1013  
  setSubStr() 1013  
  spaces() 1014  
  strLen() 1014  
  textLen() 1015  
  upperCase() 1015  
  structure-field arrays 79  
  structures 27  
  Substrings 882  
  syntax diagrams 884  
  SysLib  
  audit() 1018  
  beginDatabaseTransaction() 1020  
  bytes() 1020  
  calculateChkDigitMod10() 1021  
  calculateChkDigitMod11() 1022  
  callCmd() 1023  
  commit() 1024  
  conditionAsInt() 1024  
  connect() 1025  
  convert() 1027  
  defineDatabaseAlias() 1029  
  disconnect() 1030  
  disconnectAll() 1030  
  errorLog() 1031  
  getCmdLineArg() 1031  
  getCmdLineArgCount() 1032  
  getMessage() 1032  
  getProperty() 1033  
  loadTable() 1033  
  maximumSize() 1034  
  purge() 1035  
  queryCurrentDatabase() 1035  
  rollback() 1036  
  setCurrentDatabase() 1036  
  setError() 1037  
  setLocale() 1038  
  setRemoteUser() 1039  
  size() 1040  
  startCmd() 1040  
  startLog() 1041  
  startTransaction() 1041  
  unloadTable() 1043  
  verifyChkDigitMod10() 1043  
  verifyChkDigitMod11() 1044  
  wait() 1045  
  writeStderr() 1046  
  writeStdout() 1046  
  system build descriptor option 494  
  system libraries  
  DateTimeLib 919

- system limits 590
- system words
  - Web application 931
- systemType 1074
- SysVar
  - arrayIndex 1065
  - callConversionTable 1066
  - conversationID 1067
  - errorCode 1068
  - formConversionTable 1069
  - overflowIndicator 1069
  - returnCode 1070
  - sessionID 1071
  - sqlca 1071
  - sqlcode 1072
  - sqlState 1073
  - systemType 1074
  - terminalID 1075
  - transactionID 1075
  - transferName 1076
  - userID 1076

**T**

- tan() 984
- tanh() 984
- targetNLS build descriptor option 495
- TCP/IP listeners 436, 442
- templates, preferences 119
- terminalID 1075
- text applications
  - formGroup parts 183
  - modified data tags 191
  - segmentation 189
  - starting 424
- text expressions 601
- text forms 188
- text, preferences 115
- textLen() 1015
- textUI program parts 844
- TIME 44
- timeFormat
  - primitive field-level property 827
- timeOf() 926
- TIMESTAMP 44
- timestampFormat
  - primitive field-level property 828
- timestampFrom() 927
- timestampValue() 927
- timestampValueWithPattern() 928
- timeValue() 928
- trademarks 1225
- transactionID 1075
- transfer of control across programs 93
- transfer statement 752
- transferName 1076
- transferToProgram elements
  - alias 1089
- transferToTransaction elements
  - alias 1089
  - description 1088
  - externallyDefined 1089
- truncateBlob() 964
- truncateClob() 965
- try statement 754
- type callLink element property 515
- type definitions 28

- typeChkMsgKey
  - primitive field-level property 829
- typedefs 28

**U**

- UI parts
  - VGUI record 1089
  - VGUIRecord 167
- uiType
  - primitive field-level property 829
- UNICODE 41
- UNIX curses library 436
- UNIX users
  - ConsoleUI screen options 213
- unloadTable() 1043
- updateBlobToFile() 965
- updateClobToFile() 965
- updateWindowAttributes() 916
- upperCase
  - primitive field-level property 831
- upperCase() 1015
- use declarations 1091
- user interface (UI) parts
  - editing 195
  - form 184, 606
  - formGroup 603
  - FormGroup 183
  - page field properties 792
- userID 1076

**V**

- VAGCompatibility build descriptor
  - option 497
- validateSQLStatements build descriptor
  - option 497
- validation properties 68
- validationFailed() 918
- validationMsgNum 1061
- validationOrder
  - primitive field-level property 831
- validatorDataTable
  - primitive field-level property 832
- validatorDataTableMsgKey
  - primitive field-level property 833
- validatorFunction
  - primitive field-level property 833
- validatorFunctionMsgKey
  - primitive field-level property 834
- validValues
  - primitive field-level property 834
- validValuesMsgKey
  - primitive field-level property 836
- value
  - primitive field-level property 836
- variables, declarations 53
- variables, references to 59
- verifyChkDigitMod10() 1043
- verifyChkDigitMod11() 1044
- VGLib
  - compareBytes() 1050
  - concatenateBytes() 1052
  - connectionService() 1047
  - copyBytes() 1053
  - getVAGSysType() 1054

- VGLib (*continued*)
  - VGTDLI() 1056
- VGTDLI() 1056
- VGUI record parts 1089
- VGUIRecord record parts 167
- VGVar
  - currentFormattedGregorianCalendar 1078
  - currentFormattedJulianDate 1079
  - currentFormattedTime 1080
  - currentGregorianCalendar 1080
  - currentJulianDate 1081
  - currentShortGregorianCalendar 1081
  - currentShortJulianDate 1082
  - handleHardIOErrors 1082
  - handleOverflow 1083
  - handleSysLibraryErrors 1084
  - mqConditionCode 1084
  - sqlerrd 1085
  - sqlerrmc 1086
  - sqlIsolationLevel 1086
  - sqlWarn 1087
- VGWebTransaction program parts 847
- VisualAge Generator
  - EGL compatibility 532
  - migration from 14
- VSAM
  - access prerequisites 335
  - support 335
  - system names 335

**W**

- wait() 1045
- Web applications
  - Page Designer 221
  - support 215
- Web service definition files 15
- Web-application system words 931
- weekdayOf() 929
- What's new 1
- What's new in EGL 3
- What's new in EGL 6.0 5
- What's new in the EGL 6.0 iFix 4
- while statement 755
- Window
  - fields 553
- workbench, generation in 414, 416
- writeStderr() 1046
- writeStdout() 1046

**X**

- XML report design document 254
  - data types in 256
  - overview 251

**Y**

- yearOf() 929

**Z**

- zeroFormat
  - primitive field-level property 837









Program Number: 5724-J19

Printed in USA

SC31-6839-03

