

Rational Business Developer



Tutorial: Access relational databases with EGL and SQL

Version 7.1

Rational Business Developer



Tutorial: Access relational databases with EGL and SQL

Version 7.1

Note

Before using this information and the product it supports, read the information in "Notices," on page 65.

This edition applies to version 7.1 of Rational Business Developer and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright International Business Machines Corporation 2000, 2008. All rights reserved.**

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

Contents

Access relational databases with EGL and SQL 1

Introduction	1
Lesson 1: Set up the database connection	2
Creating an EGL project	2
Importing and connecting to the database	3
Running a test program	8
Lesson 2: Basic SQL operations in EGL	9
Reviewing the CRUD operations	11
Lesson checkpoint	17
Lesson 3: Managing a cursor with open and forEach	17
Stepping through a result set	20
Lesson checkpoint	22
Lesson 4: Constructing queries dynamically with prepare	22
Common errors in prepared statements	23
Lesson 4A: Using a prepared statement	24
Lesson 4B: Host variables in prepare statements	27
Lesson checkpoint	30
Lesson 5: Retrieving more complex data with table joins	31
Lesson 6: Executing arbitrary SQL code	35
Using the SQL builder to create SQL statements	35

Using execute	42
Lesson checkpoint	45
Lesson 7: Exception handling and rollbacks.	45
Exception handling for SQL data access	45
Rolling back changes to the database	47
Lesson checkpoint	50
Summary	50
Troubleshooting	50
Runtime error messages	53
Resources	53
Completed CRUDTest.egl file after lesson 2.	54
Completed selectItems.egl file after lesson 3	56
Completed selectItems.egl file after lesson 4A	56
Completed selectItems.egl file after lesson 4B	58
Completed printCustomerOrders.egl and records.egl files after lesson 5	60
Completed CustomerTableOperations.egl and duplicateTable.egl files after lesson 6	62
Completed exceptionHandlingTest.egl file after lesson 7.	63

Appendix. Notices	65
Trademarks	67

Access relational databases with EGL and SQL

In this tutorial, you use EGL and other tools in the workbench to work with a relational database. You learn how to use the default SQL code generated by EGL and to write your own SQL code and integrate that custom SQL with your EGL applications.

Learning objectives

This tutorial covers the use of EGL to access relational databases at an intermediate level, including the following topics:

- How to perform the four basic data access operations with EGL (create, read, update, and delete)
- How to customize the SQL code that EGL creates for you
- How to create and run customized SQL statements inside or outside EGL code
- How to handle errors in data access applications

Time required

90 minutes

Related information

Introducing EGL (a quick-start guide)

Create a hello world service with EGL

Create a hello world program with EGL

Build a JSF search page with EGL



[View the PDF version](#)

Introduction

In this tutorial, you use EGL and other tools in the workbench to work with a relational database. You learn how to use the default SQL code generated by EGL and to write your own SQL code and integrate that custom SQL with your EGL applications.

Enterprise Generation Language (EGL) is a development environment and programming language that you can use to write full-function applications quickly, freeing you to focus on the business problem your code is addressing rather than on software technologies.

Learning objectives

This tutorial covers the use of EGL to access relational databases at an intermediate level, including the following topics:

- How to perform the four basic data access operations with EGL (create, read, update, and delete)
- How to customize the SQL code that EGL creates for you
- How to create and run customized SQL statements inside or outside EGL code
- How to handle errors in data access applications

Time required

This tutorial should take approximately 90 minutes to finish. If you explore other concepts related to this tutorial, it could take longer to complete.

Skill level

Intermediate

Prerequisites

Before beginning this tutorial, you should have an intermediate knowledge of SQL and relational databases. Also, it will be easier for you to follow the EGL code examples if you have learned the basics of EGL from a tutorial such as *Introducing EGL (a quick-start guide)*.

Expected results

Optional: You could provide code snippets, screen shots, links to multimedia, or a textual description of the expected result of completing the tutorial. In other words, if there is an end product that you can show or demonstrate or describe, you could do it here, if you did not describe it earlier.

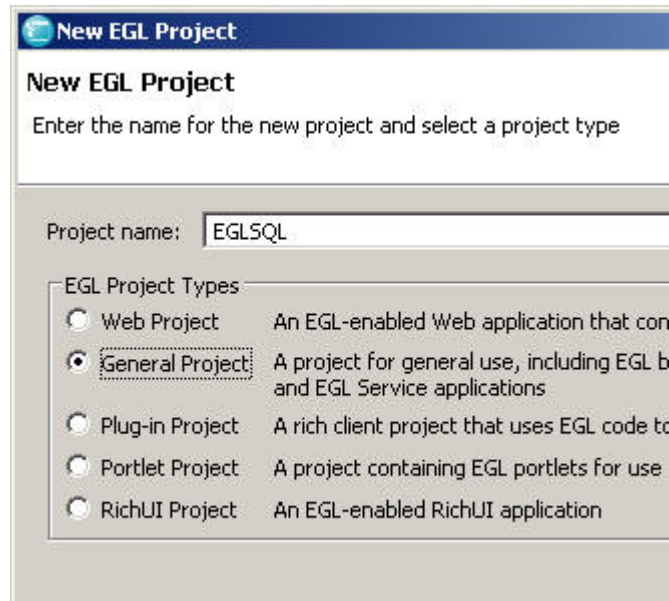
Lesson 1: Set up the database connection

The first step in accessing the database is to set up the connection. From that connection, EGL will create a starting set of data parts and logic parts, and you will write a simple data access program to test the connection.

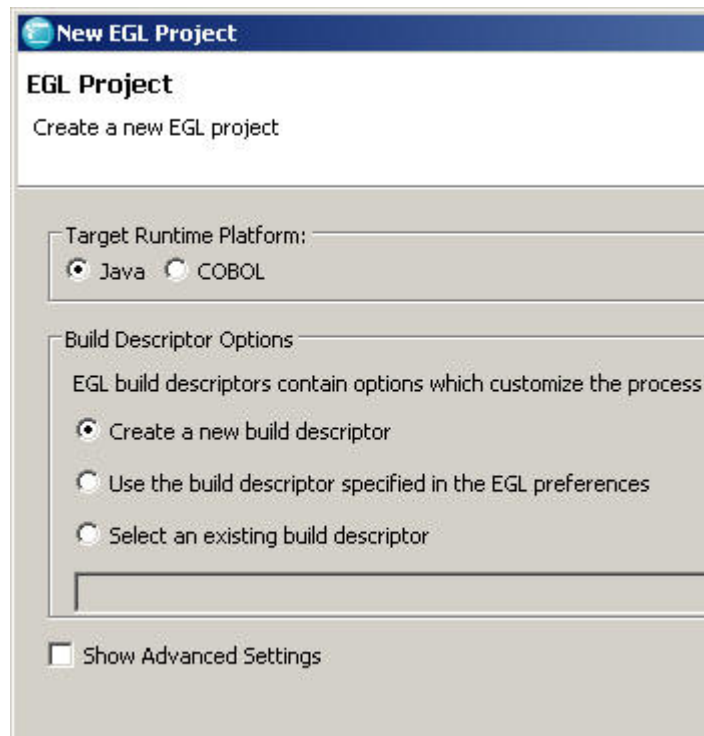
Creating an EGL project

First, you need a new EGL project to hold the data access code.

1. Switch to the EGL perspective:
 - a. Click **Window** → **Open Perspective** → **Other**.
 - b. In the Open Perspective window, click **EGL**. If you don't see EGL in the list of perspectives, select the **Show all** check box.
 - c. Click **OK**.
2. Create a new EGL project (not an EGL Web project) named EGLSQL:
 - a. Click **File** → **New** → **Project**.
 - b. In the New Project window, expand **EGL** and then click **EGL Project Wizard**.
 - c. Click **Next**.
 - d. In the **Project Name** field, enter EGLSQL.
 - e. Under **EGL Project Types**, click **General Project**. The New EGL Project window looks like this:



- f. Click **Next**.
- g. On the second page, make sure that **Java** is selected under **Target Runtime Platform** and that **Create a new build descriptor** is selected under **Build Descriptor Options**. The New EGL Project window looks like this:



- h. Click **Finish**.

The new project is created and appears in the Project Explorer view.

Importing and connecting to the database

This tutorial uses the sample Derby database that is used in the tutorial *Introducing EGL (a quick-start guide)*. In these steps, you unzip another copy of this database to

a location on your computer. You also use the Data Access Application wizard to connect to the database and create data parts and logic parts based on that database. For more information on these parts, see *Introducing EGL (a quick-start guide)*.

1. Click the following link and download the sample database to a temporary folder on your computer, such as your desktop:

Sample database

It doesn't matter where you save the database, as long as you can find it again later.

Alternately, you can find this sample database in your product installation directory in the following location:

`shared_resources/plugins/com.ibm.etools.egl.tutorial0001.doc_version/resources/EGLDerbyR7.zip`

shared_resources

The shared resources directory for your product, such as C:\Program Files\IBM\SDP70Shared on a Windows system or /opt/IBM/SDP70Shared on a Linux system. If you installed and kept a previous version of an IBM product containing EGL before installing your current product, you may need to specify the shared resources directory that was set up in the earlier install.

version

The installed version of the plugin, including three numbers separated by periods, a string separator, and the date and time that the plugin was built; for example, 7.0.0.RFB_20070120_1300. If more than one is present, use the one with the most recent version number, unless you have a reason to use an older version.

2. Unzip the database to a location on your computer, such as C:\databases. Again, it isn't important where you put the database as long as you can find it again later.
3. Click **File** → **New** → **Other**. The New window opens.
4. Expand **EGL** and click **EGL Data Access Application**.
5. Click **Next**.
6. On the Define project settings page, select **EGLSQL** in the **Project Name** list.
7. Next to the **Database Connection** list, click **New**.
8. In the New Connection window, under **Select a database manager**, expand **Derby** and click **10.1**.
9. In the **Database location** field, browse to the EGLDerbyR7 folder, which was in the ZIP file you unzipped earlier. For example, if you unzipped the EGLDerbyR7.zip file to C:\databases, the **Database location** field should be C:\databases\EGLDerbyR7

You do not need to change the **User ID** or **Password** fields.

10. In the **Class location** field, enter the path to the file derby.jar.

There are several ways you may be able to find this file:

- If you have installed IBM® WebSphere® Application Server, version 6.1, you can use the version of Derby that is included with the server. Look for derby.jar in the following folder:

`install_location/runtimes/base_v61/derby/lib`

- If you have installed database tools along with your product, you may be able to find the file in the following location:

`shared_resources/plugins/com.ibm.datatools.db2.cloudscape.driver_version/driver`

shared_resources

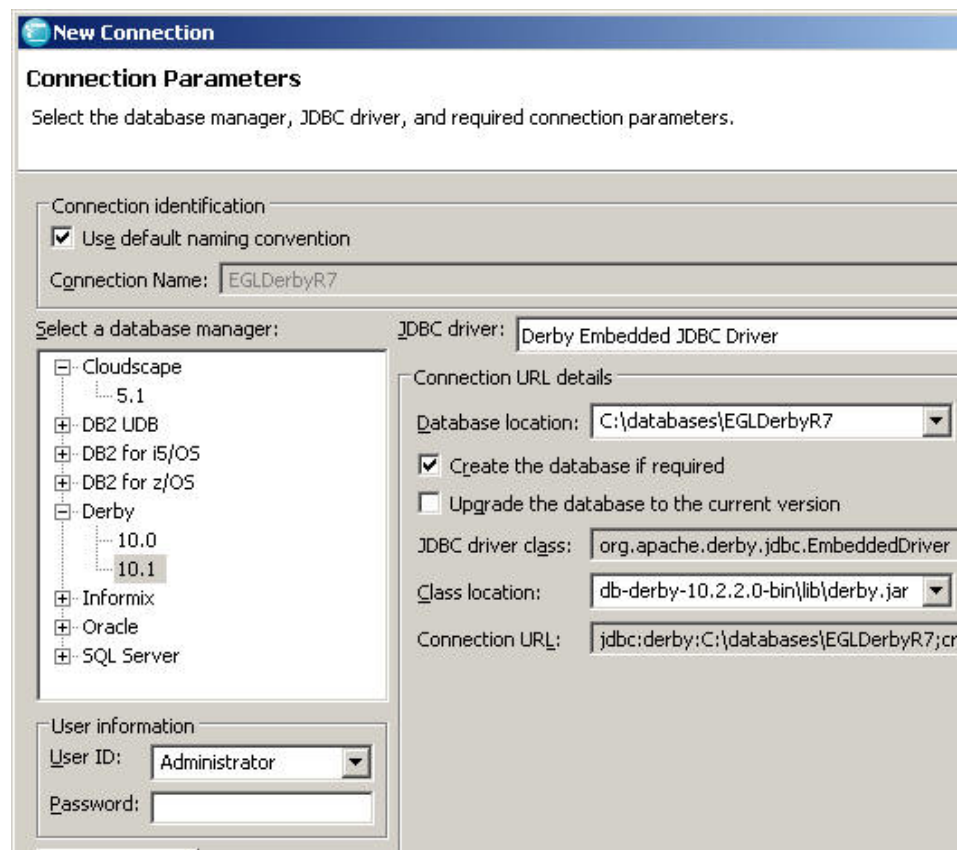
The shared resources directory for your product, such as C:\Program Files\IBM\SDP70Shared on a Windows system or /opt/IBM/SDP70Shared on a Linux system. If you installed and kept a previous version of an IBM product containing EGL before installing your current product, you may need to specify the shared resources directory that was set up in the earlier install.

version

The installed version of the plugin, including three numbers separated by periods, a string separator, and the date and time that the plugin was built; for example, 7.0.0.RFB_20070120_1300. If more than one is present, use the one with the most recent version number, unless you have a reason to use an older version.

- If you can't find the file in the previous two locations, try searching for the derby.jar file in your product installation directory. Different installations may have the file in different locations.
- If you can't find the file on your computer, you can download the file directly from the Derby Web site: <http://db.apache.org/derby/>. You will need to download the most recently released version of Derby and extract the derby.jar file to a place on your computer.

The New Connection window looks like this, with your own workspace and location information in the **Database location** and **Class location** fields:



11. Make sure that the **Use default naming convention** check box is selected.

12. Click **Finish**. Now you have established a connection to the database. All of the tables in the database are listed under **Table Name** at the bottom of the wizard. You won't create data parts for all of these tables because some contain only metadata.
13. Under **Select Tables**, select the check boxes next to only the following tables:
 - EGL.CUSTOMER
 - EGL.ITEM
 - EGL.ORDERS
 - EGL.ORDER_ITEM
 - EGL.SITEUSER
 - EGL.STATETABLE

The EGL Data Access Application wizard looks like this:

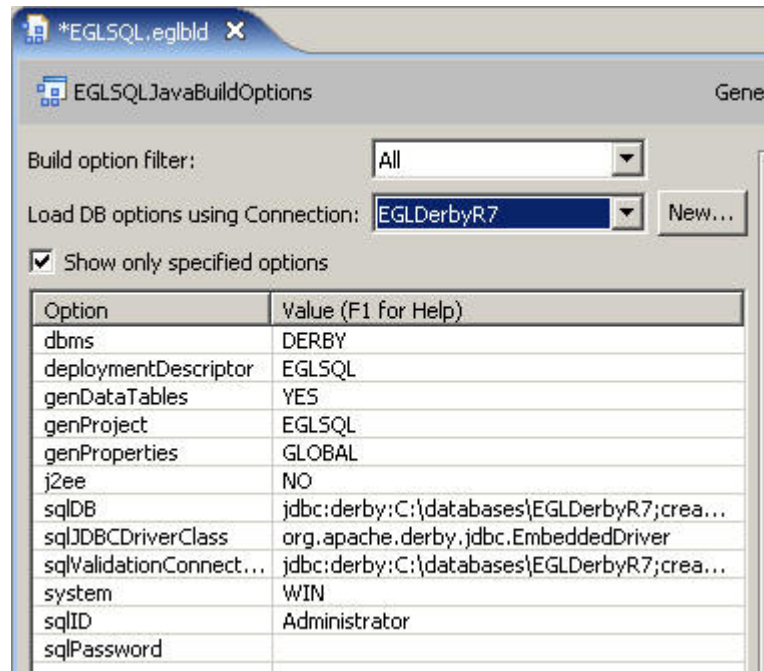
The screenshot shows the 'EGL Data Access Application' wizard window. The title bar says 'EGL Data Access Application'. The main heading is 'Define project settings' with a subtitle 'Specify project location and select tables from relational database.' Below this, there are several fields and checkboxes. 'Project Name:' is set to 'EGLSQL'. Under 'Project contents', 'Use default location' is checked, and 'Project location:' is 'D:\workspaces\SQLTest2\EGLSQL'. 'Database Connection:' is 'EGLDerbyR7'. Under 'Select Tables:', there is a table with a 'Table Name' column and checkboxes. The checked tables are EGL.CUSTOMER, EGL.ITEM, EGL.ORDERS, EGL.ORDER_ITEM, EGL.SITEUSER, and EGL.STATETABLE. The unchecked tables are EGL.SYSALIASES and SYS.SYSALIASES. At the bottom, 'Create web pages' is unchecked.

Table Name
☒ EGL.CUSTOMER
☒ EGL.ITEM
☒ EGL.ORDERS
☒ EGL.ORDER_ITEM
☒ EGL.SITEUSER
☒ EGL.STATETABLE
☐ EGL.SYSALIASES
☐ SYS.SYSALIASES

14. Note the name of the connection in the **Database Connection** field. You will need the name of the connection in a later step. Normally, the connection is named EGLDerbyR7, after the name of the database. However, if you completed the tutorial *Introducing EGL (a quick-start guide)* in this workspace, you have already created a database connection with this name, so EGL names the new connection EGLDerbyR71.
15. Clear the **Create Web pages** check box.
16. Make sure that your project **EGLWeb** is listed in the **Project Name** field.

Note: Don't click Finish yet! You need to change one more setting in this wizard.

17. Click **Next** to move to the Define the Fields page. This page lets you add key fields to the database tables. Don't change any settings on this page; the database already has a key field in each table.
18. Click **Next** again to move to the Define project creation options page.
19. On the Define project creation options page, select the **Qualify table names with schema** check box.
20. Click **Finish**. EGL creates a variety of data parts and logic parts that you can use to access the database. You will use these parts extensively in the rest of this tutorial.
You must still configure the build descriptor to use this database connection at run time.
21. Open the project's default build descriptor, which is located in EGLSQL/EGLSrc/EGLSQL.eglbld. Double-click this file to open it in the EGL Build Parts Editor.
22. In the EGL Build Parts editor, next to **Load DB options using Connection**, select the name of your database connection, which was the name of the database connection that you noted in a previous step. The build descriptor options are set to the necessary values to use that connection:

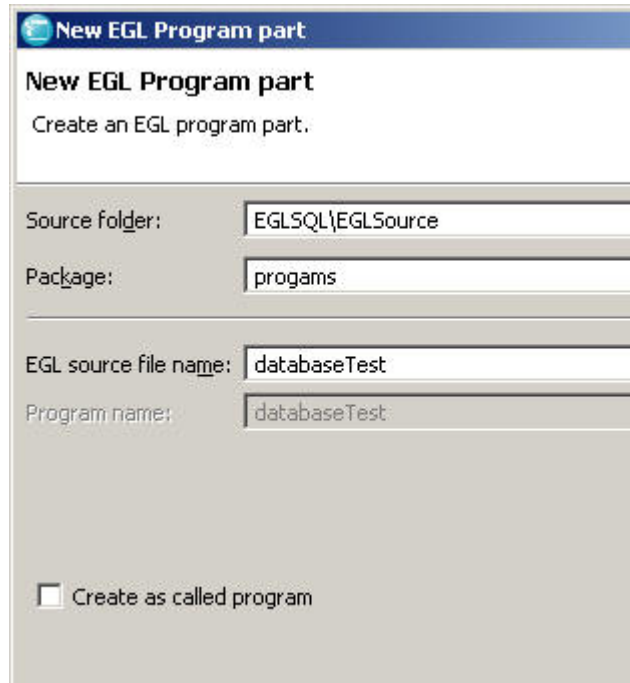


23. Save and close the build descriptor.
24. Add the Derby driver to the project's Java™ build path:
 - a. Right-click the EGLSQL project and then click **Properties**.
 - b. In the Properties window, click **Java Build Path**.
 - c. On the Java Build Path page, go to the **Libraries** tab.
 - d. On the Libraries tab, click **Add External JARs**.
 - e. In the JAR Selection window, select the Derby.jar file and click **Open**.
 - f. Click **Finish**.
 - g. Click **OK**.

Running a test program

To make sure you have the database connection working, follow these steps to run a test program and print the rows of the database to the console.

1. Click **File** → **New** → **Program**.
2. In the New EGL Program part window, type programs in the **Package** field.
3. Type databaseTest in the **EGL source file name** field. The New EGL Program part window looks like this:



4. Click **Finish**. The new program is created and opens in the editor.
5. Replace all the code in the new program with this code:

```
package programs;

import egl Derby7.data.Customer;

program databaseTest type BasicProgram {}

function main()
    customer Customer;
    open allResults for customer;

    SysLib.writeStdout("#::" Customer name");
    forEach (customer)
        SysLib.writeStdout(customer.CustomerId::" "
            ::customer.FirstName::" " ::customer.LastName);
    end
end

end
```

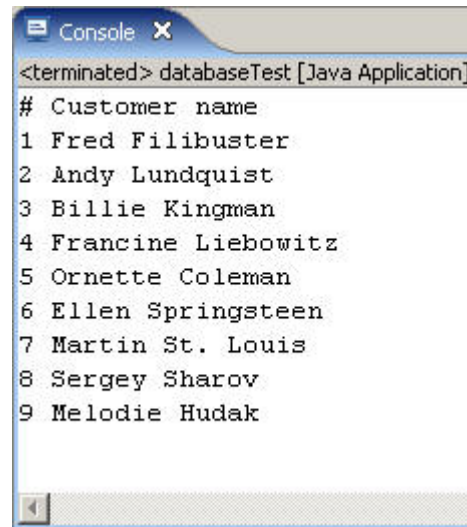
6. Save the program.
7. Generate the entire project by right-clicking the EGLSQL project in the Project Explorer view and then clicking **Generate**.

Before you can run the program, you must disconnect the database tools from the database. Derby allows only one connection at a time, and the tools EGL

used to create the data parts and set the values for the build descriptor remain connected until you close the connection.

8. Close the database connection so the database will be available to the program at run time:
 - a. Open the Database Explorer view by clicking **Window** → **Show View** → **Other**, selecting **Data** → **Database Explorer** from the list of views, and then clicking **OK**.
 - b. In the Database Explorer view, expand **Connections**. Your database connection is listed under **Connections**, with the same name as you selected in the EGL build descriptor, such as EGLDerbyR7.
 - c. Right-click your database connection and then click **Disconnect**.
 - d. Close the Database Explorer view. You will learn more about data tools, including this view, later in the tutorial.
9. Run the test program:
 - a. Expand EGLSQL/JavaSource/programs. The JavaSource folder holds the generated output of your programs.
 - b. Right-click the databaseTest.java file and then click **Run As** → **Java Application**.

When you run the test program, you should see the contents of the database's CUSTOMER table printed to the Console view:



```
<terminated> databaseTest [Java Application]
# Customer name
1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak
```

If you see an error message instead of the output in the Console view, see "Troubleshooting" on page 50.

Lesson 2: Basic SQL operations in EGL

If you went through the tutorial *Introducing EGL (a quick-start guide)*, you learned about the EGL parts necessary to access a relational database and the commands with which you can read and write data in a database. This lesson covers these concepts in more detail.

When you ran the data access application wizard, either in the previous lesson or in a past tutorial, EGL created data parts and logic parts based on the information in the database.

The most prominent data parts created are the SQLRecord parts. The fields in these Record parts relate to columns in the database. When you create a variable based on this record you can use that variable to represent a new or existing row in the database. Though each SQLRecord created by the data access application wizard matches only the columns in a single table, later in this tutorial you will learn how to merge data from more than one table into a single record (called a *table join* in SQL).

For example, you will use the Customer SQLRecord throughout this tutorial; you can find this record in the file EGLSource/eglderbyr7/data/customer.egl:

```
record Customer type sqlRecord {
  tablenames=[["EGL.CUSTOMER"]],
  keyItems=[CustomerId]
}

CustomerId CustomerId {column="EGL.CUSTOMER.CUSTOMER_ID"};
FirstName FirstName {column="EGL.CUSTOMER.FIRST_NAME",
  sqlVariableLen=yes, maxLen=30, isSqlNullable=yes};
LastName LastName {column="EGL.CUSTOMER.LAST_NAME",
  sqlVariableLen=yes, maxLen=30, isSqlNullable=yes};
...
Directions Directions {column="EGL.CUSTOMER.DIRECTIONS",
  sqlVariableLen=yes, maxLen=255, isSqlNullable=yes};
end
```

The code `tablenames=[["EGL.CUSTOMER"]]` specifies the table in the database that this record relates to, and the **column** properties on each of the fields specify the column in the table that the field relates to. The code `keyItems=[CustomerId]` indicates that the CustomerId field, and by connection, the EGL.CUSTOMER.CUSTOMER_ID column of the database, is the primary key for this table.

The data access application wizard also created several libraries that perform CRUD (create, read, update, and delete) operations on records and arrays of records. These libraries are more sophisticated ways of performing the CRUD operations directly with EGL commands. In this tutorial, you will use the EGL commands directly, but of course, it's convenient to separate logic into libraries for reuse. The four EGL CRUD commands are:

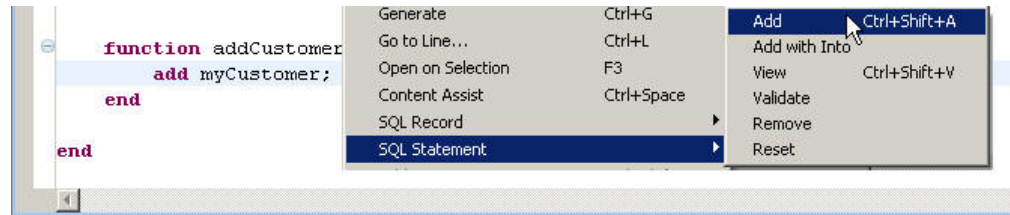
Table 1. EGL CRUD commands and their SQL equivalents

EGL command	EGL example	SQL command
add	add myRecord;	INSERT
get	get myRecord;	SELECT
replace	replace myRecord;	UPDATE
delete	delete myRecord;	DELETE

When you use one of these commands, EGL generates a default SQL statement from that command. For example, take the following **add** statement:

```
add myCustomer;
```

If you place the cursor on the myCustomer record, right-click, and then click **SQL Statement** → **Add**, the EGL editor expands the **add** statement to show the SQL code, making the SQL code *explicit*:



```

add myCustomer with
#sql{
  insert into EGL.CUSTOMER
    (EGL.CUSTOMER.CUSTOMER_ID, EGL.CUSTOMER.FIRST_NAME,
     EGL.CUSTOMER.LAST_NAME, EGL.CUSTOMER.PASSWORD,
     EGL.CUSTOMER.PHONE, EGL.CUSTOMER.EMAIL_ADDRESS,
     EGL.CUSTOMER.STREET, EGL.CUSTOMER.APARTMENT,
     EGL.CUSTOMER.CITY, EGL.CUSTOMER."STATE",
     EGL.CUSTOMER.POSTALCODE, EGL.CUSTOMER.DIRECTIONS)
  values
    (:myCustomer.CustomerId, :myCustomer.FirstName,
     :myCustomer.LastName, :myCustomer.Password,
     :myCustomer.Phone, :myCustomer.EmailAddress,
     :myCustomer.Street, :myCustomer.Apartment,
     :myCustomer.City, :myCustomer.State,
     :myCustomer.Postalcode, :myCustomer.Directions)
};

```

Nothing of substance has changed; EGL has merely exposed the SQL code that it generates from the **add** statement. Now that the code is explicit, you can edit the default SQL generated from the EGL command.

The VALUES clause of the SQL INSERT statement lists the values to insert into the new database row, in this case, fields in the myCustomer record. Each value inside the **#sql** block is preceded by a colon (:), indicating that the value is not an SQL value but an EGL value. These EGL values used in the explicit SQL code are referred to as *host variables*.

As another example of host variables at work, here is a function that uses a **get** statement to remove a series of records from a database:

```

function getCustFromState(customerState CHAR(2) in,
  customers Customer[] out)

  get customers with
  #sql{
    select *
    from EGL.CUSTOMER
    where EGL.CUSTOMER."STATE" like :customerState
  };

end

```

This function receives a two-byte CHAR parameter and uses that parameter as a host variable to retrieve the records that have a STATE field matching those two characters. Careful use of host variables is critical to making EGL work with explicit SQL code.

Reviewing the CRUD operations

In this section, you use **get**, **add**, **replace**, and **delete** to perform basic database operations with EGL. This section is optional; you can skip it if you are comfortable with the four basic EGL data access statements and their SQL equivalents.

1. Create a new EGL program named CRUDTest.egl:
 - a. In the Project Explorer view, right-click the EGLSQL project and then click **New** → **Program**. The New EGL Program part window opens.
 - b. In the **Package** field, type the name programs.
 - c. In the **EGL source file name** field, type the name CRUDTest.
 - d. Click **Finish**.

The new EGL program is created and opens in the editor.

2. Replace the default EGL code with the following code:

```
package programs;

import egl Derby7.data.*;

program CRUDTest type BasicProgram {}

function main()

  try
    // Print the starting records in the database.
    SysLib.writeStdout("Contents of database before any SQL operations:");
    printAllCustomers();

    // What to do if an error happens.
    onException (exception SQLException)
      handleDBException(exception);
    end
  end

  // This function prints out the contents of the CUSTOMER
  // table in the database.
  function printAllCustomers()

    allCustomers Customer[0];

    get allCustomers;
    for (counter int from 1 to allCustomers.getSize())
      SysLib.writeStdout(allCustomers[counter].CustomerId::" "::
        allCustomers[counter].FirstName::" "::
        allCustomers[counter].LastName);
    end
    SysLib.writeStdout("\n");
  end

  //This function responds to errors accessing the database.
  function handleDBException(exception SQLException)
    SysLib.writeStdout("Error accessing database. "::
      "See Troubleshooting section");
    SysLib.writeStdout(exception.message);
  end
end
```

Right now, this program merely prints out the complete list of customers in the database. It also includes a function named `handleDBException` to process any problems EGL encounters when working with the database. (You'll learn more about handling errors later in this tutorial.) You can run this program now to see how the database looks before you have changed it.

3. Save and generate the program.
4. Run the program:
 - a. In the Project Explorer view, expand EGLSQL/JavaSource/programs.

- b. Right-click the CRUDTest.java file and then click **Run As → Java Application**.

If the program runs without any errors, the Console view shows the contents of the database. The results in the Console view look similar to this:

Contents of database before any SQL operations:

```
1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak
```

If the results in the console include errors, make sure you have copied the program accurately, and then see “Troubleshooting” on page 50.

5. Add a function to insert a new row into the database, before the final **end** statement:

```
function addCustomer()
    customerToAdd Customer;
    customerToAdd.CustomerId = 50;
    customerToAdd.FirstName = "John";
    customerToAdd.LastName = "Doe";

    // Insert the record into the database
    try
        add customerToAdd;
        SysLib.writeStdout("Contents of database after adding a new record:");
        printAllCustomers();
    onException(exception SQLException)
        handleDBException(exception);
    end
end
```

This function uses **add** to insert a record into the database. To see the implicit SQL code behind the **add** statement, place the cursor directly on the keyword **customerToAdd** in the line **add customerToAdd;**, right-click, and then click **SQL Statement → Add**. The statement now looks like this:

```
try
    add customerToAdd with
        #sql{
            insert into EGL.CUSTOMER
                (EGL.CUSTOMER.CUSTOMER_ID, EGL.CUSTOMER.FIRST_NAME,
                 EGL.CUSTOMER.LAST_NAME, EGL.CUSTOMER.PASSWORD,
                 EGL.CUSTOMER.PHONE, EGL.CUSTOMER.EMAIL_ADDRESS,
                 EGL.CUSTOMER.STREET, EGL.CUSTOMER.APARTMENT,
                 EGL.CUSTOMER.CITY, EGL.CUSTOMER."STATE",
                 EGL.CUSTOMER.POSTALCODE, EGL.CUSTOMER.DIRECTIONS)
            values
                (:customerToAdd.CustomerId, :customerToAdd.FirstName,
                 :customerToAdd.LastName, :customerToAdd.Password,
                 :customerToAdd.Phone, :customerToAdd.EmailAddress,
                 :customerToAdd.Street, :customerToAdd.Apartment,
                 :customerToAdd.City, :customerToAdd.State,
                 :customerToAdd.Postalcode, :customerToAdd.Directions)
        };
end
```

You can also preview the SQL code without making it explicit by placing the cursor directly on the record variable, right-clicking, and clicking **SQL**

Statement → View. To work with SQL code within EGL code like this, the cursor in the EGL editor must be somewhere in the statement.

6. Call the new function from the program's main() function:

```
function main()

    // Print the starting records in the database.
    SysLib.writeStdout("Contents of database before any SQL operations:");
    printAllCustomers();

    // Add a record.
    addCustomer();

end
```

7. Save, generate, and run the program. The output shows the contents of the table before and after you add a record to it:

Contents of database before any SQL operations:

```
1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak
```

Contents of database after adding a new record:

```
1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak
50 John Doe
```

8. Add a function to delete the record from the database, before the final **end** statement:

```
function deleteCustomer()
    customerToDelete Customer;
    customerToDelete.CustomerId = 50;

    // Delete the record.
    try
        delete customerToDelete noCursor;
        SysLib.writeStdout("Contents of database after deleting the record:");
        printAllCustomers();
    onException(exception SQLException)
        handleDBException(exception);
    end

end
```

9. In the main() function, replace the call to the function that adds the record with a call to the function that deletes the record. Because the CUSTOMER_ID column is the primary key for the table, and the table already has a row with CUSTOMER_ID set to 50, trying to add the record again will cause an error.

```
function main()

    // Print the starting records in the database.
    SysLib.writeStdout("Contents of database before any SQL operations:");
```

```

printAllCustomers();

// Delete a record.
deleteCustomer();

```

```
end
```

10. Save, generate, and run the program. The output shows the contents of the table before and after you delete the record from it:

Contents of database before any SQL operations:

```

1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak
50 John Doe

```

Contents of database after deleting the record:

```

1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak

```

11. Add a function to retrieve and update the record in the database, before the final **end** statement:

```

function updateCustomer()
    customerToUpdate Customer;
    customerToUpdate.CustomerId = 50;

    // Retrieve the row.
    get customerToUpdate;

    // Change values in the record variable.
    customerToUpdate.FirstName = "Johnny";
    customerToUpdate.LastName = "Five";

    // Update the row in the database.
    try
        replace customerToUpdate noCursor;
        SysLib.writeStdout("Contents of database after updating the record:");
        printAllCustomers();
    onException(exception SQLException)
        handleDBException(exception);
    end

```

```
end
```

12. In the program's main() function, call the add, replace, and delete functions in that order:

```

function main()

    // Print the starting records in the database.
    SysLib.writeStdout("Contents of database before any SQL operations:");
    printAllCustomers();

    // Add a record.
    addCustomer();

```

```

// Update the record.
updateCustomer();

// Delete a record.
deleteCustomer();

```

end

13. Save, generate, and run the program. The output shows the contents of the table at the beginning of the program, after you add a record, after you update the record, and after you delete the record:

Contents of database before any SQL operations:

```

1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak

```

Contents of database after adding a new record:

```

1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak
50 John Doe

```

Contents of database after updating the record:

```

1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak
50 Johnny Five

```

Contents of database after deleting the record:

```

1 Fred Filibuster
2 Andy Lundquist
3 Billie Kingman
4 Francine Liebowitz
5 Ornette Coleman
6 Ellen Springsteen
7 Martin St. Louis
8 Sergey Sharov
9 Melodie Hudak

```

Here is the complete code of the CRUDTest.egl file. If you see any errors marked by red X symbols in the file, make sure your code matches the code in this file: “Completed CRUDTest.egl file after lesson 2” on page 54.

Lesson checkpoint

This lesson reviewed the basics of accessing a relational database with EGL and SQL.

You should now be familiar with the following concepts:

- SQLRecord parts and their relationship to database tables
- The four basic EGL statements that perform database operations
- Explicit SQL
- Host variables

See the following help topics for more detail on the four basic EGL data access statements:

- add
- delete
- get
- replace

Lesson 3: Managing a cursor with open and forEach

Cursors are SQL constructs that behave like a bookmark, pointing to a row and allowing you to step through a series of search results (that is, table rows) one at a time. EGL can manage cursors for you, but it also provides the **open** and **forEach** statements to take advantage of cursors.

In the previous lesson, you may have noticed that the **delete** statement included the keyword **noCursor**. Whereas the **get** statement selects one or more rows from an entire table, the **delete** statement normally operates on a row indicated by a cursor that was created by a previous statement. The **noCursor** keyword indicates that no such cursor exists and that the statement should use a WHERE clause to identify the row in the same way as the default **get** statement.

The EGL **open** statement creates a cursor and indicates a *result set*, or a set of one or more rows that the cursor can move through. Instead of accessing the cursor directly, you use other EGL statements to retrieve the next row from the result set as indicated by the cursor, or perform operations on the entire result set, using the cursor to step through the rows one at a time.

To create a cursor and result set in EGL, use the **open** statement to identify one or more rows and give the new result set a name and a temporary variable to put the row indicated by the cursor into:

```
tempItem Item;  
open allExpensiveItems for tempItem;
```

In this example, tempItem is a single record variable from the SQLRecord part representing the ITEMS table of the database. The code for tempItem specifies that EGL will use the cursor to move rows into this record one at a time. The code open allExpensiveItems gives the result set a name; allExpensiveItems is an identifier, not a variable, strictly speaking.

Like a **get** statement, EGL creates an SQL SELECT statement from the **open** statement, which you can make explicit and edit. The default SELECT statement for the **open** statement example above looks like this:

```
tempItem Item;

open expensiveItems for tempItem with
#sql{
  select
    EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
    EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
  from EGL.ITEM
  where
    EGL.ITEM.ITEM_ID >= :tempItem.ItemId
  order by
    EGL.ITEM.ITEM_ID asc
};
```

This default WHERE clause specifies only that each item must have an ITEM_ID value higher than the one before. (EGL manages the SQL DECLARE CURSOR and OPEN CURSOR statements and does not show them in the explicit SQL.)

To select particular rows to add to the result set, you must edit the WHERE clause, just like with the **get** statement. For example, you can select only the items with a cost higher than a given variable:

```
tempItem Item;
maxCost Price = 500;

open expensiveItems for tempItem with
#sql{
  select
    EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
    EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
  from EGL.ITEM
  where
    EGL.ITEM.PRICE >= :maxCost
  order by
    EGL.ITEM.ITEM_ID asc
};
```

Now the expensiveItems result set contains all of the rows from the database with a PRICE column greater than or equal to the maxCost variable, which is set to 500.

To access these rows, use the **forEach** statement, which runs a block of code in a loop, applying that code to each item in the result set. During each iteration of the loop, EGL moves the row indicated by the cursor into the temporary variable and advances the cursor to the next row in the result set. For example, the following code prints out values from each item with a price greater than 500:

```
tempItem Item;
maxCost Price = 500;

open expensiveItems for tempItem with
#sql{
  select
    EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
    EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
  from EGL.ITEM
  where
    EGL.ITEM.PRICE >= :maxCost
  order by
    EGL.ITEM.ITEM_ID asc
};

foreach (tempItem)
  SysLib.writeStdout(tempItem.Name :: " " ::
    tempItem.Description :: " $" :: tempItem.Price);
end
```

You can also use **get next** to step through the results manually. In this case you must detect whether there are any more results in the result set, such as by testing the value of **sysVar.sqlData.sqlcode**, which is set to zero as long as there are more results in the result set:

```
tempItem Item;
maxCost Price = 500;

open expensiveItems for tempItem with
#sql{
  select
    EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
    EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
  from EGL.ITEM
  where
    EGL.ITEM.PRICE >= :maxCost
  order by
    EGL.ITEM.ITEM_ID asc
};

// Retrieve the first row.
get next tempItem;

// As long as there are more rows,
// move the next row into the temporary variable
// and print its values.
while (sysvar.sqlData.sqlcode == 0)
  SysLib.writeStdout(tempItem.Name :: " " ::
    tempItem.Description :: " $" :: tempItem.Price);
  get next tempItem;
end
```

The **get** statement has a variety of other operations that you can use with an open cursor and result set:

- When using **get next**, you can specify the result set, rather than the temporary variable as in the previous example. For example, **get next from expensiveItems** moves the next row from the result set into the temporary variable; this code is equivalent to **get next tempItem**. Similarly, you can use the result set name in the **forEach** statement; **forEach (from expensiveItems)** is equivalent to **forEach (tempItem)**.
- **get current** retrieves the row currently indicated by the cursor without moving the cursor to the next row.
- **get first** and **get last** retrieve the first and last row in the results, respectively.
- **get previous** retrieves the row prior to the cursor, the converse of **get next**
- **get absolute(position)** retrieves the row at a particular position in the result set. For example, **get absolute(5) tempItem** retrieves the fifth row in the result set, and **get absolute(-2) tempItem** retrieves the second to last row.
- **get relative(position)** retrieves the row at a particular position relative to the cursor's current position. For example, **get relative(3) tempItem** retrieves the third row following the cursor's current position, and **get relative(-2) tempItem** retrieves the second row before the cursor's current position. **get relative(0) tempItem** is equivalent to **get current tempItem**.

To use these positional **get** statements other than the **get next**, you must indicate that the result set is **scrollable**, or that you intend to change the cursor's position manually, rather than step through one row at a time from beginning to end. Add the **scroll** keyword into the **open** statement as in this example:

```
open expensiveItems scroll for tempItem;
```

EGL automatically closes the cursor when it encounters any of several conditions, including a **get next** statement that returns no result because the cursor is at the end of the end of the result set, another **open** statement on the same result set, or the end of the program. If you are finished with the cursor and result set, you can close them with the EGL **close** statement, as in this example:

```
close expensiveItems;
```

Once the cursor and result set are closed, **get next** and other statements that depend on the cursor no longer work.

Stepping through a result set

In this lesson, you will step through a result set with the **forEach** statement to retrieve items from the database above a certain price.

1. Create a new program in the programs package named `selectItems`.
2. Remove the default code from the program so it looks like this:

```
package programs;

program selectItems type BasicProgram {}

    function main()
    end

end
```

3. Add a function to handle any errors EGL may encounter while accessing the database:

```
function handleDBException(exception SQLException)
    SysLib.writeStdout("Error accessing database. "::
        "See Troubleshooting section");
    SysLib.writeStdout(exception.message);
end
```

You will learn more about handling errors encountered in database access later in this tutorial.

4. Add a function to retrieve rows from the database based on a `Price` parameter for the maximum cost:

```
function printExpensiveItems(maxPrice Price in)

end
```

The `Price dataItem` part is based on the `DECIMAL` primitive type. It is defined in the file `eglderbyr7/primitivetypes/data`.

5. If you did not use content assist to add the `Price` variable in the function declaration, add the following **import** statement to the top of the file, just below the **package** statement:

```
import eglderbyr7.primitivetypes.data.*;
```

Remember that you can use content assist to complete the names of keywords and types by typing the first few letters and then pressing `Ctrl+Space`. When you insert a type with content assist, such as the `Price dataItem` part, content assist also adds the appropriate **import** statement to the file.

6. Within the new function, add an **open** statement to create a result set and cursor. You will also need a temporary `Item` variable to hold the row indicated by the cursor:

```
tempItem Item;
open expensiveItems for tempItem;
```

7. Again, if you did not use content assist to insert the Item record type, add an **import** statement to bring the Record part into scope:

```
import eglderbyr7.data.Item;
```

8. Make the SQL code explicit by placing the cursor on the **open** statement, right-clicking, and then clicking **SQL Statement** → **Add**. The **open** statement looks like this:

```
open expensiveItems for tempItem with
#sql{
  select
    EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
    EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
  from EGL.ITEM
  where
    EGL.ITEM.ITEM_ID >= :tempItem.ItemId
  order by
    EGL.ITEM.ITEM_ID asc
};
```

9. Change the WHERE clause in the explicit SQL code to select only the rows that have a EGL.ITEM.PRICE column value that is greater than or equal to the maxPrice variable passed to the function:

```
open expensiveItems for tempItem with
#sql{
  select
    EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
    EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
  from EGL.ITEM
  where
    EGL.ITEM.PRICE >= :maxPrice
  order by
    EGL.ITEM.ITEM_ID asc
};
```

Remember to include the colon before the maxPrice variable to indicate that it is an EGL variable, not an SQL keyword or value.

10. After the **open** statement, add a **forEach** loop that prints the information about the items to the console:

```
forEach (tempItem)
  SysLib.writeStdout(tempItem.Name :: " " ::
    tempItem.Description :: " $" :: tempItem.Price);
end
```

The completed function looks like this:

```
function printExpensiveItems(maxPrice Price in)
```

```
tempItem Item;
open expensiveItems for tempItem with
#sql{
  select
    EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
    EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
  from EGL.ITEM
  where
    EGL.ITEM.PRICE >= :maxPrice
  order by
    EGL.ITEM.ITEM_ID asc
};

forEach (tempItem)
  SysLib.writeStdout(tempItem.Name :: " " ::
    tempItem.Description :: " $" :: tempItem.Price);
end
```

11. Call the function from the main function and pass a value for maxPrice:

```
function main()
  try
    printExpensiveItems(200);
  onException(exception SQLException)
    handleDBException(exception);
  end
end
```

12. Generate and run the program.

The program prints out the names of the items for sale in the database with a price equal to or greater than the value passed to the function. For example, passing 200 to the function yields the following results:

```
Laptop PC Great little machine. Take it anywhere with you. $1111.11
Combination Safe Keep your valuables safe and secure. Put 'em in here. $222.22
Desktop PC Lots'a power, at the right price. Includes monitor. $888.55
```

Here is the complete code of the selectItems.egl file. If you see any errors marked by red X symbols in the file, make sure your code matches the code in this file: “Completed selectItems.egl file after lesson 3” on page 56.

Lesson checkpoint

In this lesson, you learned how to work with a result set one row at a time with **open** and **forEach**.

Working with a result set one row at a time using a cursor can be more convenient and more efficient than using **get** to retrieve the entire list of results into an array of records.

For more information, see the help topics on **open** and **forEach**.

Lesson 4: Constructing queries dynamically with prepare

The EGL **prepare** statement lets you construct SQL statements dynamically.

In the previous lesson, you created a function that selected items from a database that had a price greater than or equal to a parameter:

```
function printExpensiveItems(maxPrice Price in)

  tempItem Item;
  open expensiveItems for tempItem with
    #sql{
      select
        EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
        EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
      from EGL.ITEM
      where
        EGL.ITEM.PRICE >= :maxPrice
      order by
        EGL.ITEM.ITEM_ID asc
    };

  forEach (tempItem)
    SysLib.writeStdout(tempItem.Name :: " " ::
      tempItem.Description :: " $" :: tempItem.Price);
  end
```

What if you wanted to return items that had a price less than or equal to the parameter? Or, what if you wanted to return items between a maximum and

minimum price? Using explicit SQL in the **open** statement, you would have to code three functions, one for each possibility, because the comparisons in the WHERE clause of the SQL statement are hard-coded.

A more flexible way to create a SELECT statement is to use the EGL **prepare** statement to assemble the query at run time. When using **prepare**, you first create a STRING variable that contains the text of the SQL code:

```
selectStatement STRING =  
    "select * from EGL.ITEM where EGL.ITEM.PRICE >= 500";
```

Then you use the **prepare** statement to convert the string into an SQL statement. You must also specify a name for the statement and optionally, a record variable to indicate which SQLRecord part the statement will operate on:

```
tempItem Item;  
prepare preparedStatement from selectStatement for tempItem;
```

Then, you can use the prepared statement in place of explicit SQL in an EGL **get**, **open**, or **execute** statement (you will learn more about **execute** in a later lesson):

```
open expensiveItems for tempItem with preparedStatement;
```

Common errors in prepared statements

Creating an SQL statement from a string in this way is an advanced technique for programmers who are familiar with SQL. EGL does not check the SQL statement for accuracy at design time, so you must ensure that the string will resolve to a correct SQL statement at run time. Here is a list of common mistakes in working with prepared statements:

Host variables

You cannot use host variables in prepared statements in the same way as in explicit SQL. Take the following example:

```
tempItem Item;  
maxPrice Price = 500;  
selectStatement STRING =  
    "select * from EGL.ITEM where EGL.ITEM.PRICE >= :maxPrice";  
prepare preparedStatement from selectStatement for tempItem;
```

In this case, the actual SQL statement will include the string `:maxPrice` exactly as it is, without using it as a host variable and replacing it with the value of the `maxPrice` variable. The result is an incorrect SQL statement.

Instead, you must insert the values of the variables into the string:

```
tempItem Item;  
maxPrice Price = 500;  
selectStatement STRING =  
    "select * from EGL.ITEM where EGL.ITEM.PRICE >= " :: maxPrice;  
prepare preparedStatement from selectStatement for tempItem;
```

In this case, the value of the `maxPrice` variable is resolved when EGL creates the string, creating the following SQL statement:

```
select * from EGL.ITEM where EGL.ITEM.PRICE >= 500
```

Later in this lesson, you will learn an alternate way of using variables in prepared statements.

Spacing

Be careful to separate keywords in the SQL statement with spaces. For

example, take the following SQL statement, which is prepared by concatenating several string values with the concatenation operator (:=):

```
incorrectSQL STRING;  
incorrectSQL = "select * from EGL.ITEM where";  
incorrectSQL ::= "EGL.ITEM.PRICE >= :maxPrice";  
incorrectSQL ::= "order by EGL.ITEM.PRICE";
```

This example creates the following incorrect SQL statement:

```
select * from EGL.ITEM whereEGL.ITEM.PRICE >= :maxPriceorder by EGL.ITEM.PRICE
```

This statement needs a space after WHERE and another before ORDER BY.

Correct and appropriate statements

EGL does not validate prepared statements, so it is up to you to ensure that your statement will be correct SQL.

Also, prepared SQL statements, like explicit SQL, must be appropriate for the EGL statement with which they are used. For example, the EGL **get** and **open** statements expect a result set. To use a prepared statement with either of these statements, the SQL code must return a result set. For this reason, you cannot prepare an SQL INSERT or DELETE statement and then use that statement with **get** or **open**.

Lesson 4A: Using a prepared statement

In this lesson, you will alter the program you created in the previous exercise to accept two parameters, either or both of which can be null. The function will create a query based on which parameters are not null, returning items within a certain range of prices.

1. Open the selectItems.egl file.
2. Add a function named printItemRange that receives two nullable Price parameters:

```
function printItemRange(minPrice Price? in, maxPrice Price? in)  
  
end
```

The question mark (?) after the parameter type indicates that the variable can receive a null value. This way, the function will be able to accept either or both of a minimum and maximum for the range of results.

3. Create a STRING variable to hold the SQL code and insert the beginning of a SELECT statement into it:

```
selectStatement STRING = "select * from EGL.ITEM where ";
```

The next step is to determine what the appropriate WHERE clause should be. There are four possible situations:

- Both parameters are null. In this case, the function will print all of the items with a price greater than or equal to zero, so the WHERE clause should be where EGL.ITEM.PRICE >= 0.
- Both parameters are specified. In this case, the function will use the SQL BETWEEN keyword to print values between or equal to the parameters. The WHERE clause should be where EGL.ITEM.PRICE between :minPrice and :maxPrice.
- Only the maximum parameter is specified. In this case, the function will print all of the items with a price less than or equal to the maximum: where EGL.ITEM.PRICE <= :maxPrice.

- Only the minimum parameter is specified. In this case, the function will print all of the items with a price greater than or equal to the maximum: where `EGL.ITEM.PRICE >= :minPrice`.

There are several ways to make this decision in EGL, but a simple way is to use three nested `if` statements.

4. Add the following code to the function to complete the `WHERE` clause:

```
if (minPrice == NULL && maxPrice == NULL)
    // Two empty parameters;
    // return all rows with PRICE greater than zero.
    selectStatement ::= "EGL.ITEM.PRICE > 0 ";
else
    if (minPrice != NULL && maxPrice != NULL)
        // Both parameters were specified.
        selectStatement ::= "EGL.ITEM.PRICE between " :: minPrice :: " and " :: maxPrice :: " ";
    else
        // Only one parameter was specified.
        if (minPrice == NULL)
            // Maximum was specified.
            selectStatement ::= "EGL.ITEM.PRICE <= " :: maxPrice :: " ";
        else
            // Minimum was specified.
            selectStatement ::= "EGL.ITEM.PRICE >= " :: minPrice :: " ";
        end
    end
end
```

5. Complete the statement with an `ORDER BY` clause:

```
selectStatement ::= "order by EGL.ITEM.PRICE";
```

6. To make sure the SQL statement is correct, print it to the console:

```
SysLib.writeStdout("Completed query: "::selectStatement);
```

7. Prepare the statement to be used, referring to a record created from the table that the prepared statement will access:

```
tempItem Item;
prepare preparedStatement
from selectStatement
for tempItem;
```

8. Use the prepared statement in place of explicit SQL in an **open** statement:

```
open expensiveItems for tempItem with preparedStatement;
```

9. Print the results of the query:

```
forEach (tempItem)
    SysLib.writeStdout(tempItem.Name :: " " ::
        tempItem.Description :: " $" :: tempItem.Price);
end
```

The completed function looks like this:

```
function printItemRange(minPrice Price? in, maxPrice Price? in)

    selectStatement STRING = "select * from EGL.ITEM where ";

    if (minPrice == NULL && maxPrice == NULL)
        // Two empty parameters;
        // return all rows with PRICE greater than zero.
        selectStatement ::= "EGL.ITEM.PRICE > 0 ";
    else

        if (minPrice != NULL && maxPrice != NULL)
            // Both parameters were specified.
            selectStatement ::= "EGL.ITEM.PRICE between " :: minPrice :: " and " :: maxPrice :: "
```

```

else
    // Only one parameter was specified.
    if (minPrice == NULL)
        // Maximum was specified.
        selectStatement ::= "EGL.ITEM.PRICE <= " :: maxPrice :: " ";
    else
        // Minimum was specified.
        selectStatement ::= "EGL.ITEM.PRICE >= " :: minPrice :: " ";
    end
end

end

// Regardless of parameters, sort by price.
selectStatement ::= "order by EGL.ITEM.PRICE";

// Print completed SQL code.
SysLib.writeStdout("Completed query: "::selectStatement);

// Prepare the statement to be used.
tempItem Item;
prepare preparedStatement
    from selectStatement
    for tempItem;

// Use the prepared statement in place of explicit SQL.
open expensiveItems for tempItem with preparedStatement;

// Print the results to the console.
forEach (tempItem)
    SysLib.writeStdout(tempItem.Name :: " " ::
        tempItem.Description :: " $" :: tempItem.Price);
end

end

```

10. Call the function from the program's main function:

```

function main()
    minPrice, maxPrice Price?;
    minPrice = 50;
    maxPrice = 500;
    try
        printItemRange(minPrice, maxPrice);
    onException(exception SQLException)
        handleDBException(exception);
    end
end

```

11. Generate and run the program.

With the parameter values in the example above, the function prints the items with a price between 50 and 500:

```

Completed query: select * from EGL.ITEM where EGL.ITEM.PRICE
    between 50.00 and 500.00 order by EGL.ITEM.PRICE
Serial Mouse Great little mouse. All kinds'a applications. $65.22
Keyboard Got QWERTY? If not, go get this awesome flat keyboard. $78.99
Watch Keep time - and look rakish! $99.01
Shredder Don't let that politically-sensitive document fall into the wrong hands! $198.99
Combination Safe Keep your valuables safe and secure. Put 'em in here. $222.22

```

You can edit the parameter values and try different ranges, including null values, as in this example:

```

function main()
    minPrice, maxPrice Price?;
    minPrice = 500;
    maxPrice = null;

```

```

try
    printItemRange(minPrice, maxPrice);
onException(exception SQLException)
    handleDBException(exception);
end
end

```

Passing values of 500 and null yields results like these:

```

Completed query: select * from EGL.ITEM where
EGL.ITEM.PRICE >= 500.00 order by EGL.ITEM.PRICE
Desktop PC Lots'a power, at the right price. Includes monitor. $888.55
Laptop PC Great little machine. Take it anywhere with you. $1111.11

```

Here is the complete code of the selectItems.egl file. If you see any errors marked by red X symbols in the file, make sure your code matches the code in this file: "Completed selectItems.egl file after lesson 4A" on page 56.

Lesson 4B: Host variables in prepare statements

The SQL statement in the previous example was limited in that the variables in the WHERE clause were fixed. It's possible to create a prepared statement before you know which variables will be used in the statement. This type of statement is more complicated, but it can provide flexibility if you want to prepare a statement and fill in the details later.

In the previous example, you had to resolve the values of the variables in order to concatenate them with the rest of the query string:

```

selectStatement STRING = "select * from EGL.ITEM where ";
...
selectStatement ::= "EGL.ITEM.PRICE between " :: minPrice :: " and " :: maxPrice :: " ";

```

When you create a prepared statement, you can also place question marks (?) into the statement, indicating that the values will be filled in later:

```

selectStatement2 STRING = "select * from EGL.ITEM where " ::
"EGL.ITEM.PRICE between ? and ? " ::
"order by EGL.ITEM.PRICE";

```

```

tempItem Item;
prepare preparedStatement2
from selectStatement2
for tempItem;

```

Now the prepared statement has places for two host variables, represented by the question marks in the BETWEEN clause. When you are ready to use the prepared statement, insert values with the **using** statement. For example, to search for rows with a price between 5 and 500, pass the literals 5 and 500 to the statement:

```

open expensiveItems2 for tempItem with preparedStatement2
using 5, 500;

```

Of course, you can also use variables along with the **using** keyword:

```

open expensiveItems2 for tempItem with preparedStatement2
using minPrice, maxPrice;

```

Using host variables in a prepared statement in this way can be useful if you want to assemble a statement once and call it several times with different values. For example, this function uses the same prepared statement three times to print values within three different ranges:

```

function printRanges()

    tempItem Item;
    selectStatement string = "select * from EGL.ITEM where " ::
        "EGL.ITEM.PRICE between ? and ? " ::
        "order by EGL.ITEM.PRICE";

    prepare preparedStatement from selectStatement for tempItem;

    SysLib.writeStdout("\nPrinting values between 0 and 200");
    open expensiveItems for tempItem with preparedStatement using 0, 200;
    foreach(tempItem)
        SysLib.writeStdout(tempItem.Name :: " " :: tempItem.Description ::
            " $" :: tempItem.Price);
    end

    SysLib.writeStdout("\nPrinting values between 200 and 500");
    open expensiveItems for tempItem with preparedStatement using 200, 500;
    foreach(tempItem)
        SysLib.writeStdout(tempItem.Name :: " " :: tempItem.Description ::
            " $" :: tempItem.Price);
    end

    SysLib.writeStdout("\nPrinting values between 500 and 1,000");
    open expensiveItems for tempItem with preparedStatement
        using 500, 1000;
    foreach(tempItem)
        SysLib.writeStdout(tempItem.Name :: " " :: tempItem.Description ::
            " $" :: tempItem.Price);
    end

end

```

Using host variables in this way, it's possible to rewrite the program from the previous section to use a single prepared statement, instead of altering the statement for each possible case:

1. In the selectItems.egl file, add a new function that accepts a maximum and minimum parameter, just like in the previous section:

```

function printItemRange2(minPrice Price? in, maxPrice Price? in)

end

```

In this function, you will prepare the statement first and insert variables into it later.

2. Create a string variable to hold the prepared statement and then use the **prepare** statement to prepare it:

```

selectStatement STRING = "select * from EGL.ITEM where " ::
    "EGL.ITEM.PRICE between ? and ? " ::
    "order by EGL.ITEM.PRICE";

tempItem Item;
prepare preparedStatement
    from selectStatement
    for tempItem;

```

You will pass two parameters to this prepared statement, representing the bounds of the search. If the maximum price is not specified, you will need to know the upper boundary to search for, since you cannot convert the EGL.ITEM.PRICE between ? and ? clause into EGL.ITEM.PRICE >= ? once you have prepared the statement. Therefore, you need to determine the highest price in the table to use as the upper boundary. You can determine this upper boundary easily with the SQL MAX() function.

3. Create an Item variable and retrieve the price of the most expensive item in the database, using the MAX() function to place it into the Price field of the record variable:

```
get mostExpensiveItem
into mostExpensiveItem.Price
with
  #sql{
    select
      max(EGL.ITEM.PRICE)
    from EGL.ITEM
  };
```

Now the mostExpensiveItem record contains the highest price in the Items table.

4. Determine which parameters were specified and use the prepared statement accordingly:

```
if (minPrice == NULL && maxPrice == NULL)
  // Two empty parameters;
  open expensiveItems for tempItem with preparedStatement
  using 0, mostExpensiveItem.Price;
else
  if (minPrice != NULL && maxPrice != NULL)
    // Both parameters were specified.
    open expensiveItems for tempItem with preparedStatement
    using minPrice, maxPrice;
  else
    // Only one parameter was specified.
    if (minPrice == NULL)
      // Maximum was specified.
      open expensiveItems for tempItem with preparedStatement
      using 0, maxPrice;
    else
      // Minimum was specified.
      open expensiveItems for tempItem with preparedStatement
      using minPrice, mostExpensiveItem.Price;
    end
  end
end
```

5. Use **forEach** to print the results:

```
forEach (tempItem)
  SysLib.writeStdout(tempItem.Name :: " " ::
    tempItem.Description :: " $" :: tempItem.Price);
end
```

The completed function looks like this:

```
function printItemRange2(minPrice Price? in, maxPrice Price? in)

  selectStatement STRING = "select * from EGL.ITEM where " ::
    "EGL.ITEM.PRICE between ? and ? " ::
    "order by EGL.ITEM.PRICE";

  // Prepare the statement to be used.
  tempItem Item;
  prepare preparedStatement
  from selectStatement
  for tempItem;

  mostExpensiveItem Item;
  get mostExpensiveItem
  into mostExpensiveItem.Price
  with
```

```

        #sql{
            select
                max(EGL.ITEM.PRICE)
            from EGL.ITEM
        };

if (minPrice == NULL && maxPrice == NULL)
    // Two empty parameters;
    open expensiveItems for tempItem with preparedStatement
        using 0, mostExpensiveItem.Price;
else
    if (minPrice != NULL && maxPrice != NULL)
        // Both parameters were specified.
        open expensiveItems for tempItem with preparedStatement
            using minPrice, maxPrice;
    else
        // Only one parameter was specified.
        if (minPrice == NULL)
            // Maximum was specified.
            open expensiveItems for tempItem with preparedStatement
                using 0, maxPrice;
        else
            // Minimum was specified.
            open expensiveItems for tempItem with preparedStatement
                using minPrice, mostExpensiveItem.Price;
        end
    end
end

// Print the results to the console.
forEach (tempItem)
    SysLib.writeStdout(tempItem.Name :: " " ::
        tempItem.Description :: " $" :: tempItem.Price);
end

end

```

6. Modify the main function to call the new function instead of the old function:

```

function main()
    minPrice, maxPrice Price?;
    minPrice = 50;
    maxPrice = 500;
    try
        printItemRange2(minPrice, maxPrice);
    onException(exception SQLException)
        handledBException(exception);
    end
end

```

7. Save, generate, and run the new program with different values of minPrice and maxPrice, including null values. The results are the same as the previous function, but the function arrived at the results in a different way.

Here is the complete code of the selectItems.egl file. If you see any errors marked by red X symbols in the file, make sure your code matches the code in this file: “Completed selectItems.egl file after lesson 4B” on page 58.

Lesson checkpoint

Creating valid prepared statements can be complicated, but the resulting statements are more flexible than statements hard-coded in explicit SQL. In a later lesson, you will learn to set the default SQL code for a SQLRecord part, which offers another way of customizing SQL statements.

For more information on prepared statements, see prepare.

Lesson 5: Retrieving more complex data with table joins

Up to this point, each data access function has retrieved rows from only one table. In this lesson, you learn to use SQL table joins with EGL to work with multiple tables at once.

A full discussion of SQL table joins is beyond the scope of this tutorial. In short, a table join assembles a result set from two or more related tables.

For example, the CUSTOMERS table in the database includes a customer ID number as a primary key, along with the first and last names of the customers (as well as several other columns):

Table 2. Sample data from the CUSTOMER table

CUSTOMER_ID	FIRST_NAME	LAST_NAME
1	Fred	Filibuster
2	Andy	Lundquist
3	Billie	Kingman

The ORDERS table includes an order ID number as a primary key, and the amount of the order. The table also includes the ID number of the customer who placed the order, making that column a *foreign key*:

Table 3. Sample data from the ORDERS table

ORDER_ID	CUSTOMER_ID	ORDER_AMOUNT
1	1	111.11
2	1	222.22
3	1	333.33

In this case, the CUSTOMER_ID column is a good candidate for a table join because this column creates a relationship between the two tables. In this lesson, you will join these two tables with a customized SQLRecord part, producing a result set that matches the customers with the orders placed by those customers.

Table joins are a complex database operation, so you should be familiar with the behavior of the database and test your output carefully. As you'll see in this lesson, you don't always get the data you may expect.

1. Create a new EGL source file to hold a new SQLRecord part:
 - a. Right-click the EGLSQL project in the Project Explorer view and then click **New → EGL Source File**.
 - b. In the New EGL Source File window, type the name data in the **Package** field.
 - c. In the **EGL Source File Name** field, type records.
 - d. Click **Finish**.

The new EGL source file is created and opens in the editor.

2. In the new file, below the **package** statement, type in the name of a new SQLRecord to combine both the CUSTOMER and ORDERS tables:

```
record OrderCustomerJoin type SQLRecord
{tableNames = [{"EGL.CUSTOMER", "C"}, {"EGL.ORDERS", "O"}]}
end
```

The first line of this record is the same as the other SQLRecord parts: it specifies a name for the record and the SQLRecord stereotype. The second line specifies the tables to which this record relates, as values of the **tableNames** property. The single-table records you have worked with in previous lessons listed only one table in this property, such as the ORDERS table for the Orders record:

```
record Orders type sqlRecord {
  tableNames=["EGL.ORDERS"]
  ...
}
```

The OrderCustomerJoin record you are creating must list two tables in the **tableNames** property to retrieve data from both tables. For clarity, the record includes an alias for each table name ("O" for ORDERS and "C" for CUSTOMERS). These aliases make it easier to refer to the table names in the record, as you will see soon.

Now that you have specified the basic information for the Record part, EGL can fill in the rest. However, you must first specify which database EGL will retrieve connection information from.

3. Set the Derby database as the default SQL database for EGL:
 - a. Click **Window** → **Preferences**.
 - b. In the Preferences window, expand **EGL** and click **SQL Database Connections**.
 - c. In the **Connections** list, select your database connection, named EGLDerbyR7 or EGLDerbyR71.
 - d. Click **OK**.
4. Back in the records.egl file, place the cursor on the name of the record.
5. With the cursor on the name of the OrderCustomerJoin record, right-click and then click **SQL Record** → **Retrieve SQL**. The EGL SQL Retrieve feature
6. In the password prompt, leave the default values for your user name and password. The sample database does not require a particular user name or password. The EGL SQL Retrieve feature creates the rest of the SQLRecord part for you, including fields for the rows in the tables and the **keyItems** property:

```
record OrderCustomerJoin type SQLRecord
{tableNames = ["EGL.CUSTOMER", "C"], ["EGL.ORDERS", "O"]},
keyItems=[CUSTOMER_ID, ORDER_ID]}

CUSTOMER_ID int
{column="C.CUSTOMER_ID", isReadOnly=yes};
FIRST_NAME string
{column="C.FIRST_NAME", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=30};
LAST_NAME string
{column="C.LAST_NAME", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=30};
...
ORDER_ID int
{column="O.ORDER_ID", isReadOnly=yes};
orders_CUSTOMER_ID int
{column="O.CUSTOMER_ID", isReadOnly=yes,
 isSqlNullable=yes};
ORDER_AMOUNT decimal(8,2)
{column="O.ORDER_AMOUNT", isReadOnly=yes,
 isSqlNullable=yes};
ORDER_DETAILS string
{column="O.ORDER_DETAILS", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=111};
end
```

Note that the value of the **column** property for each field begins with the table's alias as specified in the **tableNames** property. This alias can help you keep track of which table a particular field relates to.

7. Save the records.egl file, but keep it open.
8. Create a new Program part in the programs package named printCustomerOrders.
9. In the new program, remove the default code.
10. In the program's main function, create an array variable based on the OrderCustomerJoin record that you just created:

```
allCustomerOrders OrderCustomerJoin[0];
```

Remember to use content assist to insert the Record part type by pressing Ctrl+Space. If you do not use content assist, you must manually add an **import** statement to the top of the file, immediately below the **package** statement:

```
import data.OrderCustomerJoin;
```

11. Use the **open** and **forEach** statements to retrieve the rows from the database and print the values to the console:

```
tempRec OrderCustomerJoin;
open resultSet for tempRec;
forEach (tempRec)
    SysLib.writeStdout(tempRec.CUSTOMER_ID :: " " ::
        tempRec.LAST_NAME :: " " :: tempRec.ORDER_ID);
end
```

12. Save, generate, and run the program. If you're familiar with the pitfalls of table joins, you may have noticed the mistake in this table join and why the output is not very useful:

```
1 Filibuster 1
1 Filibuster 2
1 Filibuster 3
...
1 Filibuster 22
1 Filibuster 23
2 Lundquist 1
2 Lundquist 2
2 Lundquist 3
2 Lundquist 4
...
2 Lundquist 22
2 Lundquist 23
3 Kingman 1
3 Kingman 2
...
```

The database has not made a logical cross-reference between the tables; instead, it has paired each row in the CUSTOMERS table with each row in the ORDERS table in every possible combination. In these results, each customer is recorded as making every order, which is obviously incorrect. If you make the SQL code behind the **open** statement explicit, you can see that the SQL is selecting every row from both tables without using a WHERE clause:

```
open resultSet for tempRec with
#sql{
    select
        C.CUSTOMER_ID, C.FIRST_NAME, C.LAST_NAME, C.PASSWORD,
        C.PHONE, C.EMAIL_ADDRESS, C.STREET, C.APARTMENT,
        C.CITY, C.STATE, C.POSTALCODE, C.DIRECTIONS,
        O.ORDER_ID, O.CUSTOMER_ID, O.ORDER_AMOUNT,
        O.ORDER_DETAILS, O.ORDER_DATE, O.ORDER_STATUS
```

```

from EGL.CUSTOMER C, EGL.ORDERS O
order by
  C.CUSTOMER_ID, O.ORDER_ID asc
};

```

To join the tables meaningfully, you must tell the database how to merge the tables logically, instead of returning the union of the tables. You could do this by editing the explicit SQL code, but in this case you would have to do this every time you use the record, or else you risk retrieving inaccurate data. Instead, you will set the default selection condition for this record so the implicit SQL code will always preform a meaningful table join.

13. If you made the SQL code behind the **open** statement explicit, make it implicit again by placing the cursor on the record name, right-clicking, and then clicking **SQL Statement** → **Remove**.
14. Return to the Record part definition in the records.egl file.
15. Add the **defaultSelectCondition** property to the Record part definition:

```

record OrderCustomerJoin type SQLRecord
{tableNames = [{"EGL.CUSTOMER", "C"}, {"EGL.ORDERS", "O"}],
keyItems=[CUSTOMER_ID, ORDER_ID],
defaultSelectCondition = #sqlCondition{ condition }}

```

The **defaultSelectCondition** property specifies the WHERE clause in the implicit SQL code and default explicit SQL code when you use **get** or **open** with this record.

16. Inside the **#sqlCondition{}** block, replacing the default "condition" text, enter the correct WHERE clause for an SQL table join between these two tables, taking care to use the table aliases instead of the full table names:

```

defaultSelectCondition = #sqlCondition{ O.CUSTOMER_ID = C.CUSTOMER_ID }

```

You do not need to add the keyword WHERE; EGL inserts it automatically in the SQL code in this case.

17. Save the file. Now if you make the SQL code behind the **open** statement explicit, it looks like this:

```

open resultSet for tempRec with
#sql{
  select
    C.CUSTOMER_ID, C.FIRST_NAME, C.LAST_NAME, C.PASSWORD,
    C.PHONE, C.EMAIL_ADDRESS, C.STREET, C.APARTMENT,
    C.CITY, C.STATE, C.POSTALCODE, C.DIRECTIONS,
    O.ORDER_ID, O.CUSTOMER_ID, O.ORDER_AMOUNT,
    O.ORDER_DETAILS, O.ORDER_DATE, O.ORDER_STATUS
  from EGL.CUSTOMER C, EGL.ORDERS O
  where
    O.CUSTOMER_ID = C.CUSTOMER_ID
  order by
    C.CUSTOMER_ID, O.ORDER_ID asc
};

```

Now the SQL code includes a WHERE clause that will limit the rows returned to those that have the same customer number in both the CUSTOMER and ORDERS table.

18. Save, generate, and run the program.

With a meaningful table join, the results of this program show which customer made each order:

```

1 Filibuster 1
1 Filibuster 2
1 Filibuster 3

```

```

...
1 Filibuster 11
1 Filibuster 18
2 Lundquist 7
2 Lundquist 8
2 Lundquist 12
2 Lundquist 15
2 Lundquist 20
3 Kingman 13
4 Liebowitz 14
6 Springsteen 22
7 St. Louis 16
8 Sharov 17
9 Hudak 23

```

Here is the complete code of the two files used in this lesson. If you see any errors marked by red X symbols in either file, make sure your code matches this code: “Completed printCustomerOrders.egl and records.egl files after lesson 5” on page 60.

Lesson 6: Executing arbitrary SQL code

In this lesson, you learn more flexible ways of working with SQL code in EGL.

Up to this point, you have been limited to using SQL statements that match an EGL statement. For example, you can edit the SQL code associated with a **get** statement, but the SQL code is limited to a **SELECT** statement. You cannot, for example, put a SQL **DELETE** statement in the explicit SQL code for a **get** statement. Even though you can edit the explicit SQL, the SQL code must still be appropriate for the EGL keyword.

Then how do you use the SQL statements that have no direct EGL analogue? You can execute SQL code that is not associated with an EGL keyword either directly through the workbench tooling, or with the EGL **execute** statement.

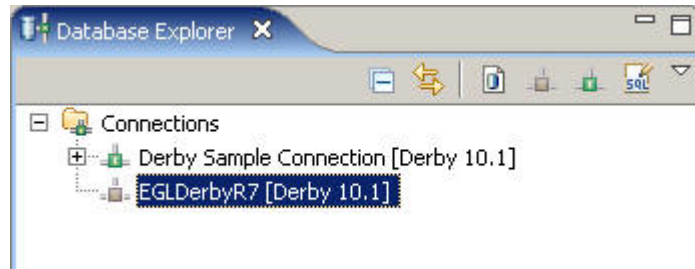
Using the SQL builder to create SQL statements

The workbench includes several tools to help you work with databases outside of an EGL application. In this section, you work with the sample database more directly with the SQL builder.

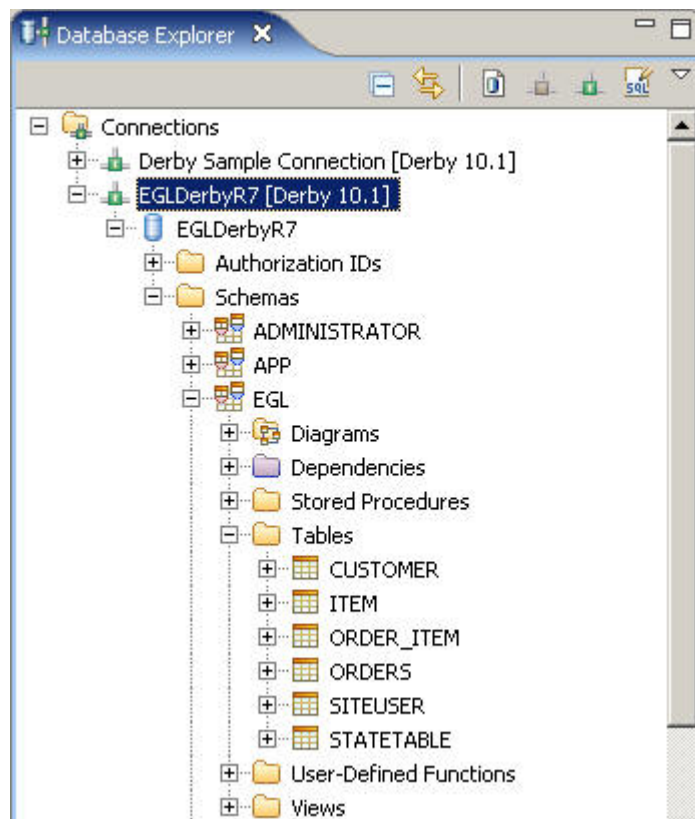
1. Switch to the Data perspective:
 - a. Click **Window** → **Open Perspective** → **Other**.
 - b. In the Open Perspective window, click **Data**.
 - c. Click **OK**.

The Data perspective opens, which includes views used for working with databases. You will be working mostly with the Database Explorer view.

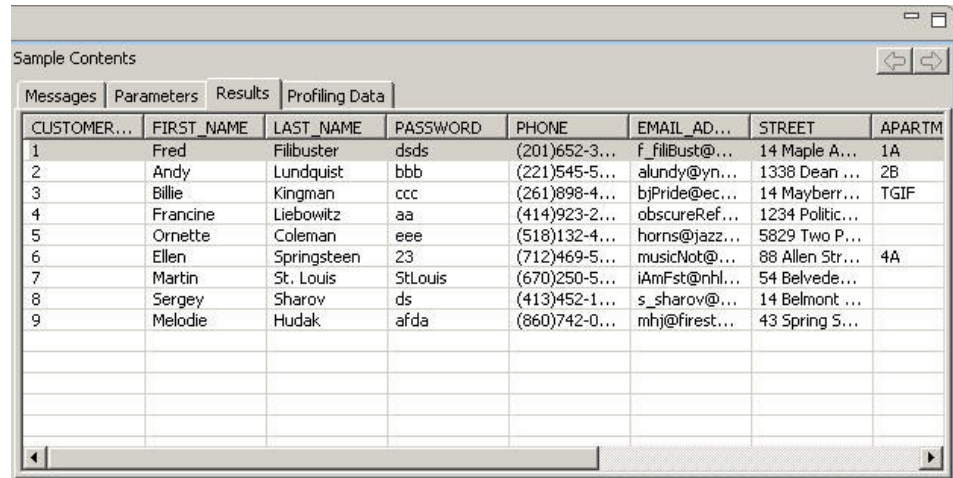
2. In the Database Explorer view, expand **Connections**. The Database Explorer view shows your project’s connection to the database as EGLDerbyR7, along with a sample Derby connection used for testing:



3. Right-click the EGLDerbyR7 connection and then click **Reconnect**. Now that the Database Explorer view is connected to the database, you can work with the database using the tools in the Data perspective. However, since Derby only allows one connection to a database at a time, as long as the Database Explorer view is connected, your EGL programs will not be able to connect to the database.
4. In the Database Authorization window, click **OK**.
5. Expand **EGLDerbyR7** → **EGLDerbyR7** → **Schemas** → **EGL** → **Tables**. Now the Database Explorer view lists the tables in the database that this tutorial is concerned with. The other tables are metadata and are not important for this tutorial.



6. Right-click the **CUSTOMERS** table and then click **Data** → **Sample Contents**. The Data Output view shows the data in the **CUSTOMERS** table:

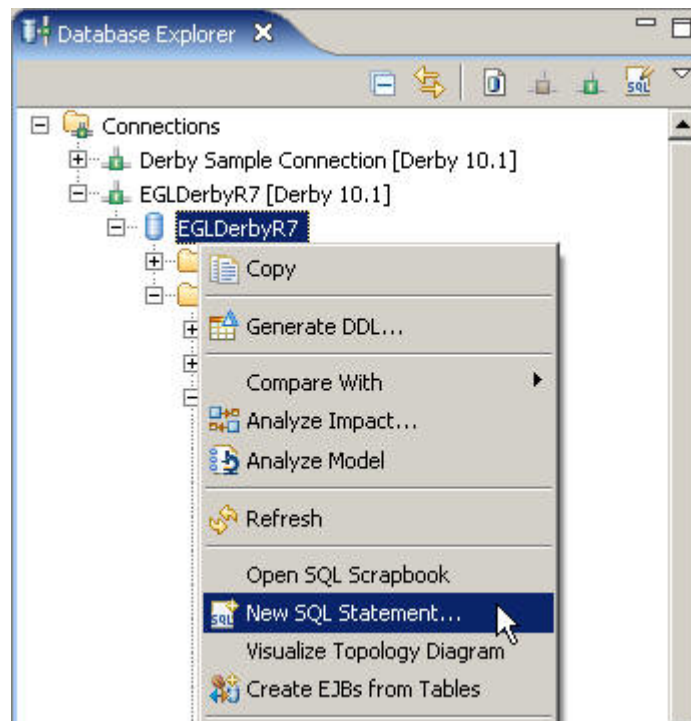


The screenshot shows a window titled 'Sample Contents' with tabs for 'Messages', 'Parameters', 'Results', and 'Profiling Data'. The 'Results' tab is active, displaying a table with 9 rows of customer data. The columns are: CUSTOMER..., FIRST_NAME, LAST_NAME, PASSWORD, PHONE, EMAIL_AD..., STREET, and APARTM.

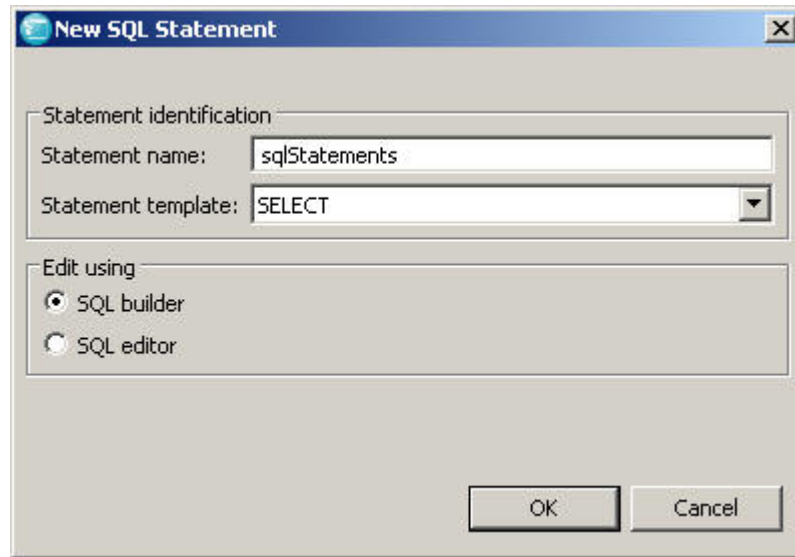
CUSTOMER...	FIRST_NAME	LAST_NAME	PASSWORD	PHONE	EMAIL_AD...	STREET	APARTM
1	Fred	Filibuster	dsds	(201)652-3...	f_filibust@...	14 Maple A...	1A
2	Andy	Lundquist	bbb	(221)545-5...	alundy@yn...	1338 Dean ...	2B
3	Billie	Kingman	ccc	(261)898-4...	bjPride@ec...	14 Mayberr...	TGIF
4	Francine	Liebowitz	aa	(414)923-2...	obscureRef...	1234 Politic...	
5	Ornette	Coleman	eee	(518)132-4...	horns@jazz...	5829 Two P...	
6	Ellen	Springsteen	23	(712)469-5...	musicNot@...	88 Allen Str...	4A
7	Martin	St. Louis	StLouis	(670)250-5...	iAmFst@nhl...	54 Belvede...	
8	Sergey	Sharov	ds	(413)452-1...	s_sharov@...	14 Belmont ...	
9	Melodie	Hudak	afda	(860)742-0...	mhj@firest...	43 Spring S...	

The Data Output view has run a default SELECT statement, like the implicit SELECT statement behind an EGL **get** statement. To work with the database more directly, and to test SQL code before adding it to EGL programs, you can use the SQL Builder to write and run SQL commands interactively.

7. In the Database Explorer view, right-click the EGLDerbyR7 database, represented by the blue database icon and then click **New SQL Statement**.



8. In the New SQL Statement window, give the statement the name sqlStatements.
9. Leave the **Statement Template** field set to SELECT and the **Edit Using** radio button group set to **SQL builder**.

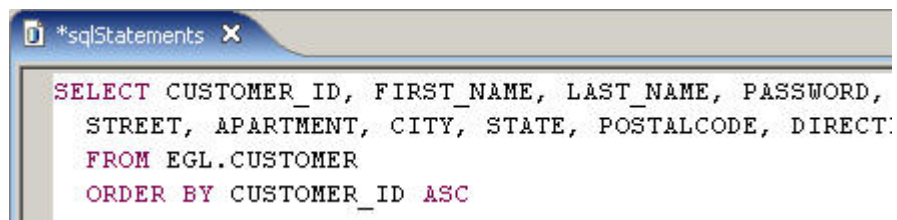


10. Click **OK**. The SQL builder opens. From this editor, you can create SQL statements and test them before you add them to your application.
11. As an example, paste the default SQL statement for the Customer record into the SQL builder. You can copy this statement (or any other SQL statement) from explicit SQL in the EGL editor, or use the following code:

```
select
  EGL.CUSTOMER.CUSTOMER_ID, EGL.CUSTOMER.FIRST_NAME,
  EGL.CUSTOMER.LAST_NAME, EGL.CUSTOMER.PASSWORD,
  EGL.CUSTOMER.PHONE, EGL.CUSTOMER.EMAIL_ADDRESS,
  EGL.CUSTOMER.STREET, EGL.CUSTOMER.APARTMENT,
  EGL.CUSTOMER.CITY, EGL.CUSTOMER."STATE",
  EGL.CUSTOMER.POSTALCODE, EGL.CUSTOMER.DIRECTIONS
from EGL.CUSTOMER
order by
  EGL.CUSTOMER.CUSTOMER_ID asc
```

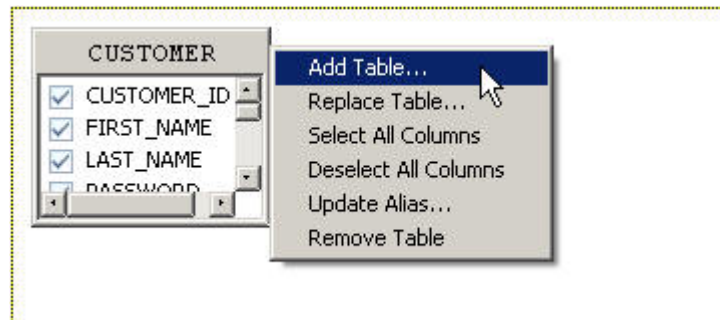
The SQL builder shows the SQL code in various different ways:

- The code editor at the top of the SQL builder allows you to edit SQL statements just like in the EGL editor. However this editor has extra options for dealing with SQL statements; for example, when you paste the default SQL statement from the EGL code into this editor, the SQL builder simplifies and reformats it:



Also, this SQL editor provides content assist for SQL statements. Just like EGL content assist, you can press Ctrl+Space for a list of suggestions.

- The second section from the top shows a graphical view of the tables involved in the statement and their columns. You can select or remove columns from the SELECT statement by clearing the check boxes next to the columns, and you can add a new table to the statement by right-clicking the editor area and then clicking **Add Table**. You can also see a list of options for a table by right-clicking the table:



- The section at the bottom shows a tabular view of the statement. You can use the templates in the tables to guide you through editing the statement.

Columns	Conditions	Groups	Group Conditions
Column	Alias	Output	Sort Type
CUSTOMER.CUSTOMER_ID		☒	
CUSTOMER.FIRST_NAME		☒	
CUSTOMER.LAST_NAME		☒	
CUSTOMER.PASSWORD		☒	

For example, you can use the tabular area of the SQL builder to add a WHERE clause to the SELECT statement in the editor:

12. Go to the **Conditions** tab of the SQL builder.
13. On the Conditions tab, add a new WHERE clause to the statement:
 - a. On the Conditions tab, click an empty row in the **Column** column.
 - b. In the blank cell under **Column**, select CUSTOMER.POSTALCODE.
 - c. Under **Operator**, select **LIKE**.
 - d. Under **Value**, type the following code:


```
1%
```

The percent symbol (%) is a wildcard in SQL. This WHERE clause limits the statement to customers with a postal code that begins with the number 1.

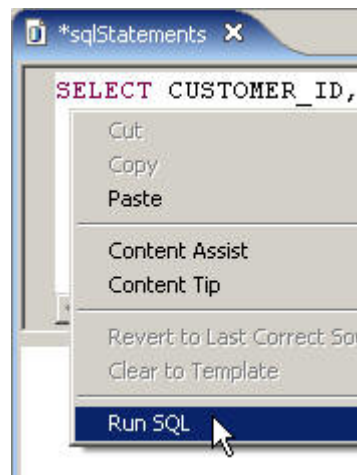
The selection condition looks like this in the tabular area:

Columns	Conditions	Groups	Group Conditions
Column	Operator	Value	AND/OR
POSTALCODE	LIKE	'1%'	

When you are finished adding the condition and set the focus elsewhere, the SQL builder updates the code of the SELECT statement in the editor:

```
*sqlStatements x
SELECT CUSTOMER_ID, FIRST_NAME, LAST_NAME,
       STREET, APARTMENT, CITY, STATE, POSTALCOI
FROM EGL.CUSTOMER
WHERE POSTALCODE LIKE '1%'
```

14. Right-click the code editor at the top of the SQL builder and then click **Run SQL**.

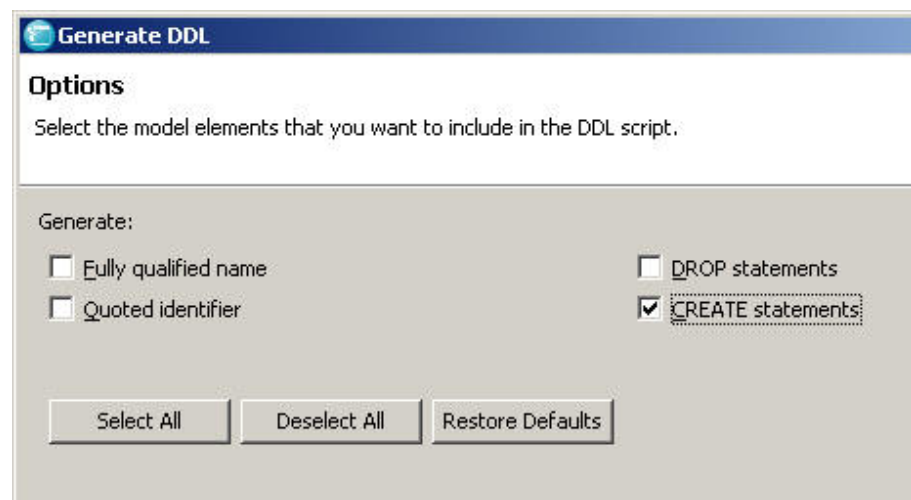


The Data Output view shows the results of the statement. In this case, it shows the customers with a postal code beginning with 1:

Messages	Parameters	Results	Profiling Data	
CUSTOMER...	FIRST_NAME	LAST_NAME	PASSWORD	PHONE
4	Francine	Liebowitz	aa	(414)923-2...
8	Sergey	Sharov	ds	(413)452-1...

If you are familiar with SQL, you can write your own SQL statements here and test them before using them in an application.

15. When you are finished with the SQL builder, you can save the code to a file or discard it.
16. As a more powerful example, create an SQL statement that will make a new table by copying the design for the CUSTOMER table:
 - a. In the Database Explorer view, right-click the CUSTOMER table and then click **Generate DDL**.
 - b. In the Generate DDL window, select the **CREATE statements** check box and clear the other check boxes.



- c. Click **Next**,

- d. On the **Objects** page, click **Select All**.
- e. Click **Next**.
- f. Select your EGLSQL project in the **Folder** field.
- g. On the Save and Run DDL file page, name the file createtable.sql.
- h. Clear the **Run DDL on server** check box.
- i. Select the **Open DDL file for editing** check box. The Save and Run DDL page looks like this:

Generate DDL

Save and Run DDL

Specify a path to save the generated DDL script. You connection information.

Folder:

File name:

Preview DDL

```
CREATE TABLE CUSTOMER (
  CUSTOMER_ID INTEGER NOT NULL,
  FIRST_NAME VARCHAR(30),
  LAST_NAME VARCHAR(30),
  PASSWORD CHAR(8),
  PHONE VARCHAR(14),
  EMAIL_ADDRESS VARCHAR(50),
  STREET VARCHAR(30),
```

Statement terminator:

☐ Run DDL on server

☒ Open DDL file for editing

- j. Click **Next**.
- k. Click **Finish**.

The new SQL statement opens in the SQL statement editor, which works like the SQL builder but has only the code editing area and not the additional graphical areas.

17. Edit the statement to change the name of the table that is created to EGL.CUSTOMERTEMP rather than CUSTOMER. The edited SQL code looks like this:

```
CREATE TABLE EGL.CUSTOMERTEMP (
  CUSTOMER_ID INTEGER NOT NULL,
  FIRST_NAME VARCHAR(30),
  LAST_NAME VARCHAR(30),
  PASSWORD CHAR(8),
  PHONE VARCHAR(14),
  EMAIL_ADDRESS VARCHAR(50),
  STREET VARCHAR(30),
  APARTMENT VARCHAR(10),
  CITY VARCHAR(30),
```

```

        STATE CHAR(2),
        POSTALCODE VARCHAR(10),
        DIRECTIONS VARCHAR(255)
    );

```

18. Save the statement.
19. Disconnect from the database by right-clicking the EGLDerbyR7 database in the Database Explorer view and then clicking **Disconnect**.

Now you have a SQL statement that will create a new table for the database. However, EGL does not have a statement that is equivalent to the SQL CREATE statement, and you cannot insert this statement into the explicit SQL for a statement such as **get** because **get** expects a result set from the database. In the next section, you will use **execute** to run this SQL statement in EGL.

Using execute

You could run the new CREATE statement using the tools in the Data perspective, but sometimes you may need to run custom SQL code in your EGL application. Obviously, any time you insert customized SQL code into an EGL application, you should test it carefully to make sure it does what you expect it to do.

The EGL **execute** statement lets you run SQL code without dealing with SQLRecord variables. You can use **execute** to run a variety of SQL statements, including CREATE, ALTER, and DROP TABLE, INSERT, DELETE, and UPDATE. For example, you can remove a table from the database like this:

```

execute #sql{
    DROP TABLE OLD_TABLE;
}

```

The **execute** statement doesn't support some SQL statements, most notably SELECT and OPEN. For a list of which SQL statements work or do not work with **execute**, see execute considerations for SQL.

You can also use **execute** to call a stored procedure:

```

execute #sql{
    call aStoredProcedure( :parameterVar)
};

```

Derby does not support stored procedures in the same way as most database management systems, so this tutorial will not cover stored procedures. Instead, in this section, you will use **execute** to create a table in the database and populate it with data without using any SQLRecord parts.

1. Return to the EGL perspective.
2. Create a new Library part in a file named CustomerTableOperations.egl in a package named libraries.
3. Insert new functions into the library named execCreateTable, execInsertIntoTable, and execDropTable, with each function receiving a StatusRec record variable as a parameter:


```

function execCreateTable(status StatusRec inOut)

end

function execInsertIntoTable(status StatusRec inOut)

end

```

```
function execDropTable(status StatusRec inOut)

end
```

The StatusRec record variable records information about the success or failure of database operations.

4. If you did not use content assist to add the parameter, add the following **import** statement to the top of the file, just below the **package** statement:

```
import egllderbyr7.StatusRec;
```

5. Using an **execute** statement in the execCreateTable function, run the CREATE TABLE statement you created in the previous section:

```
function execCreateTable(status StatusRec inOut)
try
    execute
    #sql{
        CREATE TABLE EGL.CUSTOMERTEMP (
            CUSTOMER_ID INTEGER NOT NULL,
            FIRST_NAME VARCHAR(30),
            LAST_NAME VARCHAR(30),
            PASSWORD CHAR(8),
            PHONE VARCHAR(14),
            EMAIL_ADDRESS VARCHAR(50),
            STREET VARCHAR(30),
            APARTMENT VARCHAR(10),
            CITY VARCHAR(30),
            STATE CHAR(2),
            POSTALCODE VARCHAR(10),
            DIRECTIONS VARCHAR(255)
        )
    };
    onException (exception SQLException)
        ConditionHandlingLib.HandleException(status, exception);
end

end
```

This function creates the table and then handles any errors using the HandleSuccess function that is created along with the data parts and logic parts from the database.

6. If you did not use content assist to add this code, add the following **import** statement to the top of the file:

```
import egllderbyr7.ConditionHandlingLib;
```

7. In the execDropTable function, add an **execute** statement that removes the table from the database:

```
function execDropTable(status StatusRec inOut)
try
    execute #sql{
        DROP TABLE EGL.CUSTOMERTEMP
    } ;
    onException (exception SQLException)
        ConditionHandlingLib.HandleException(status, exception);
end

end
```

8. In the execInsertIntoTable function, add an **execute** statement that moves records from the CUSTOMER table into the CUSTOMERTEMP table:

```
function execInsertIntoTable(status StatusRec inOut)
try
    execute #sql{
        INSERT INTO EGL.CUSTOMERTEMP
```

```

        SELECT * FROM EGL.CUSTOMER
      } ;
    onException (exception SQLException)
      ConditionHandlingLib.HandleException(status, exception);
  end
end

```

This code merely copies each record from CUSTOMER into CUSTOMERTEMP. If you want, you can add a WHERE clause to select only certain records to copy.

9. Save and generate the library.
10. Create a new Program part in a file named duplicateTable.egl in the programs package.
11. In the new program, call the functions that create the new table and insert records into it:


```

package programs;

import eglderbyr7.StatusRec;
import libraries.CustomerTableOperations;

program duplicateTable type BasicProgram {}

  status StatusRec;

  function main()
    CustomerTableOperations.execCreateTable(status);
    CustomerTableOperations.execInsertIntoTable(status);
  end

end

```
12. Generate and run the program.
13. Return to the Data perspective and reconnect to the database in the Database Explorer view.
14. As you did in the previous section, preview the data in the new table to see the new table and its data:
 - a. Expand **EGLDerbyR7** → **EGLDerbyR7** → **Schemas** → **EGL** → **Tables**.
 - b. Right-click the CUSTOMERTEMP table and then click **Data** → **Sample Contents**.

Now you can see the data in the new table:

Messages	Parameters	Results	Profiling Data	
CUSTOMER...	FIRST_NAME	LAST_NAME	PASSWORD	PHONE
1	Fred	Filibuster	dsds	(201)652-3...
2	Andy	Lundquist	bbb	(221)545-5...
3	Billie	Kingman	ccc	(261)898-4...
4	Francine	Liebowitz	aa	(414)923-2...
5	Ornette	Coleman	eee	(518)132-4...
6	Ellen	Springsteen	23	(712)469-5...
7	Martin	St. Louis	StLouis	(670)250-5...
8	Sergey	Sharov	ds	(413)452-1...
9	Melodie	Hudak	afda	(860)742-0...

Now you can create other statements to work with the new table, or you can call the execDropTable function to remove it from the database. For example, you might execute the ALTER TABLE and CREATE INDEX statements that were

created along with the CREATE TABLE statement, but if you do, you must rename the constraint and index, because Derby does not allow duplicate names for primary key constraints. Remember to disconnect from the database in the Database Explorer view before running any EGL code that accesses the database.

Here is the complete code of the two files used in this lesson. If you see any errors marked by red X symbols in either file, make sure your code matches this code: “Completed CustomerTableOperations.egl and duplicateTable.egl files after lesson 6” on page 62.

Lesson checkpoint

In this lesson, you learned about some of the database management tools in the workbench and how you can combine those tools with the **execute** statement to run a wide variety of SQL statements within an EGL application.

For more detail on the data access tools in the workbench, see Working with SQL statements. For more information on the EGL **execute** statement, see `execute`.

Lesson 7: Exception handling and rollbacks

It’s especially important to properly handle errors in applications that access data sources in order to avoid retrieving incorrect data, inserting incorrect data, or otherwise creating inconsistencies between the application and the data source. In this lesson, you will learn about a few tools that you can use to respond to errors when accessing relational databases.

Exception handling for SQL data access

You’ve probably noticed that the examples in this tutorial often put data access statements within a **try** block.

```
function execDropTable(status StatusRec inOut)
  try
    execute #sql{
      DROP TABLE EGL.CUSTOMERTEMP
    };
    onException (exception SQLException)
      ConditionHandlingLib.HandleException(status, exception);
  end
end
```

When you put code in a **try** block, EGL attempts to run that code as usual. If EGL encounters an error, it stops running the code in the **try** block and moves directly to the **onException** block, skipping any statements that remain in the **try** block. You can put code in the **onException** block to recover from the error, correct the problem, or write error information to the log file.

When EGL encounters an error within a **try** block, it also creates an *exception record* to provide information about the error. It passes that record to the **onException** block. Within the **onException** block, you can use the information in that record to help determine the cause of the error and recover from it. In the above example, the **onException** block passes the exception record to a function that deals with the error.

Exception records can have one of several stereotypes, depending on the type of error. In the example above, the **onException** block expects an exception record with the `SQLException` stereotype, which is for SQL errors and other relational database errors. If EGL encounters a different type of error, such as reference to a

null value, it creates a different type of exception record (in the case of a reference to a null value, an exception record with the `NullValueException` stereotype).

You can place multiple **onException** blocks after a single **try** block in order to process different types of errors differently, as in the following example:

```
function execDropTable(status StatusRec inOut)
  try
    execute #sql{
      DROP TABLE EGL.CUSTOMERTEMP
    };
    onException (exception SQLException)
      ConditionHandlingLib.HandleException(status, exception);
    onException (exception AnyException)
      HandleOtherError(exception);
  end
end
```

In this example, the first **onException** block expects a `SQLException` record and the second block expects the `AnyException` stereotype. In this case, if EGL encounters an SQL error, the first **onException** block runs; if EGL encounters any other kind of error, the second block runs. The `AnyException` stereotype is a general-user exception record stereotype, used to respond to any error, regardless of stereotype.

All exception records have at least two fields: a `messageID` field with the error code for the error and a `message` field with a brief explanation of the problem. Depending on the stereotype, exception records can have other fields. For example, the `SQLRecord` stereotype has a `sqlState` field that holds a `CHAR(5)` value with the status of the statement, and a `sqlCode` field that holds an `INT` return code from the DBMS.

You can try out exception handling by passing invalid information, deleting or updating a record that doesn't exist, or trying to add a record with the same primary key. The following example attempts to add a new customer record with the same ID number as an existing row in the database, which will cause a conflict with primary keys and prompt EGL to throw an `SQLException` exception.

1. Create a new program in the programs package with the name `exceptionHandlingTest`.
2. In the main function of the new program, create a customer record variable and assign it information including an ID number already existing in the database:

```
package programs;

import eglderbyr7.data.Customer;

program exceptionHandlingTest type BasicProgram {}

  function main()
    customer Customer;
    customer.FirstName = "John";
    customer.LastName = "Doe";
    customer.CustomerId = 1;
  end

end

3. Within a try block, add the record to the database:

  try
    add customer;
    SysLib.writeStdout("Customer added successfully.");
  end
```

This **try** block isn't very useful because there is no matching **onException** block to respond to any errors.

4. Before the **end** that closes the **try** block, add an **onException** block to handle the `SQLException` that will occur when EGL attempts to run the code in the **try** block:

```
try
  add customer;
  SysLib.writeStdout("Customer added successfully.");
onException(ex SQLException)
  SysLib.writeStdout("SQL exception " :: ex.messageID);
  SysLib.writeStdout("message = " :: ex.message);
  SysLib.writeStdout("SQLCode = " :: ex.sqlCode);
  SysLib.writeStdout("SQLState = " :: ex.sqlState);
end
```

The complete program looks like this:

```
package programs;

import eglDerby7.data.Customer;

program exceptionHandlingTest type BasicProgram {}

function main()
  customer Customer;
  customer.FirstName = "John";
  customer.LastName = "Doe";
  customer.CustomerId = 1;

  try
    add customer;
    SysLib.writeStdout("Customer and order added successfully.");
  onException(ex SQLException)
    SysLib.writeStdout("SQL exception " :: ex.messageID);
    SysLib.writeStdout("message = " :: ex.message);
    SysLib.writeStdout("SQLCode = " :: ex.sqlCode);
    SysLib.writeStdout("SQLState = " :: ex.sqlState);
  end

end

end
```

5. Save, generate, and run the program. The console shows the output from the **onException** block:

```
SQL exception EGL0504E
message = EGL0504E ADD: The statement was aborted
because it would have caused a duplicate key value
in a unique or primary key constraint or unique
index identified by 'SQL070307095259210' defined
on 'CUSTOMER'.[sqlstate:23505][sqlcode:20000]
EGL0002I The error occurred in the
exceptionHandlingTest program processing the main function.
SQLCode = 20000
SQLState = 23505
```

Rolling back changes to the database

Most relational databases, including Derby, are *recoverable* resources. When you make a change to a recoverable resource, that change is not permanent until you issue a *commit*, which saves the changes. In this way, you can reverse changes to the database if you encounter an exception before you commit those changes.

You can set the **sqlCommitControl** build descriptor option to NOAUTOCOMMIT, which prevents EGL from committing each change automatically. With automatic commits turned off, you can manually commit the changes by invoking **sysLib.commit()** and you can reverse changes made since the last commit by invoking **sysLib.rollback()**. If you do not commit the changes with **sysLib.commit()**, EGL always commits changes at the end of the run unit, such as at the end of a main program.

For example, imagine that you are making several changes to the database, such as adding a new customer and entering order information for that customer's first order. If some of these operations succeed before another fails, you may be left with a customer without any orders, or an order without a customer who made the order. In this case, you can put a rollback in the **onException** block to reverse the changes and keep incomplete information out of the database:

1. Set the **sqlCommitControl** build descriptor option to NOAUTOCOMMIT:
 - a. In the Project Explorer view, double-click the EGLSQL project's build descriptor to open it in the EGL Build Parts Editor. This file is located at EGLSQL/EGLSource/EGLSQL.eglbd.
 - b. In the Build Parts Editor, clear the **Show only specified options** check box.
 - c. Under **Option**, find **sqlCommitControl**.
 - d. Set the value for **sqlCommitControl** to NOAUTOCOMMIT.

Note: To open the **sqlCommitControl** option for editing, click twice slowly in the **Value** column next to that option. Also, you can click three times quickly in the **Value** column.

- e. Save and close the build descriptor.
2. Open the exceptionHandlingTest program that you created in the previous section.
3. Add an order record to the customer record that is already in the program:

```
program exceptionHandlingTest type BasicProgram {}

function main()
  customer Customer;
  customer.FirstName = "John";
  customer.LastName = "Doe";
  customer.CustomerId = 1;

  customerOrder Orders;
  customerOrder.CustomerId = 1;
  customerOrder.OrderId = 50;
  customerOrder.OrderAmount = 0;
  customerOrder.OrderDetails =
    "This is my first order, so get it right!";

  ...

end
```

4. Within a **try** block, add the order first and then the customer:

```
try
  add customerOrder;
  SysLib.writeStdout("Order added successfully.");
  add customer;
  SysLib.writeStdout("Customer added successfully.");
end
```

In this case, the order will be added but the customer will not, because of the duplicate primary key. Therefore, you can reverse the change to the ORDERS table by invoking a rollback in the **onException** block.

5. Add an **onException** block, including a rollback, after the **try** block:

```
try
  add customerOrder;
  SysLib.writeStdout("Order added successfully.");
  add customer;
  SysLib.writeStdout("Customer added successfully.");
onException(ex SQLException)
  SysLib.writeStdout("SQL exception " & ex.messageID);
  SysLib.writeStdout("message = " & ex.message);
  SysLib.writeStdout("Performing rollback.");
  SysLib.rollback();
end
```

Now, if either **add** statement fails, the code **onException** block runs and attempts to reverse the changes to the database.

6. After the **onException** block, add code to test whether the rollback successfully removed the order that was added before the exception occurred. To see if the order record is still in the database, you can compare it to **noRecordFound**:

```
tempOrder Orders;
tempOrder.OrderId = 50;
get tempOrder;

if (tempOrder is noRecordFound)
  SysLib.writeStdout("The order was rolled back successfully.");
else
  SysLib.writeStdout("The order is still in the database");
end
```

Another way to find out the status of a call to a database is to test the values of the **sqlLib.sqlData** system variable. This system variable is a record, and its fields store information about the success or failure of the last SQL database operation. Specifically, the field **sqlLib.sqlData.sqlcode** contains 0 if the operation completed successfully and returned results, and it contains 100 if the operation completed successfully but the SELECT statement returned no results. Therefore, you could also tell if the row was found or not with the following code:

```
if (sqlLib.sqlData.sqlcode == 0)
  SysLib.writeStdout("The order is still in the database");
else
  if (sqlLib.sqlData.sqlcode == 100)
    SysLib.writeStdout("The order was rolled back successfully.");
  end
end
```

7. Generate and run the program.

The output of the program shows that the order is added to the database, the exception occurs, and then the order is successfully removed as a result of the rollback:

```
Order added successfully.
SQL exception EGL0504E
message = EGL0504E ADD: The statement was aborted
because it would have caused a duplicate
key value in a unique or primary key constraint
or unique index identified by 'SQL070307095259210'
defined on 'CUSTOMER'.[sqlstate:23505][sqlcode:20000]
```

EGL0002I The error occurred in the exceptionHandlingTest
program processing the main function.
Performing rollback.
The order was rolled back successfully.

Here is the complete code of the exceptionHandlingTest.egl file. If you see any errors marked by red X symbols in the file, make sure your code matches the code in this file: “Completed exceptionHandlingTest.egl file after lesson 7” on page 63.

Lesson checkpoint

In this lesson, you learned the basics of handling errors in SQL operations. It’s very important to anticipate problems and provide exception handling to allow the application to recover.

For more information on the runtime errors that can cause exceptions, see EGL Java runtime error codes. For a list of the exception records that EGL can create, see EGL core Exception records.

Summary

Lessons learned

By completing this tutorial, you learned about the following concepts and tasks:

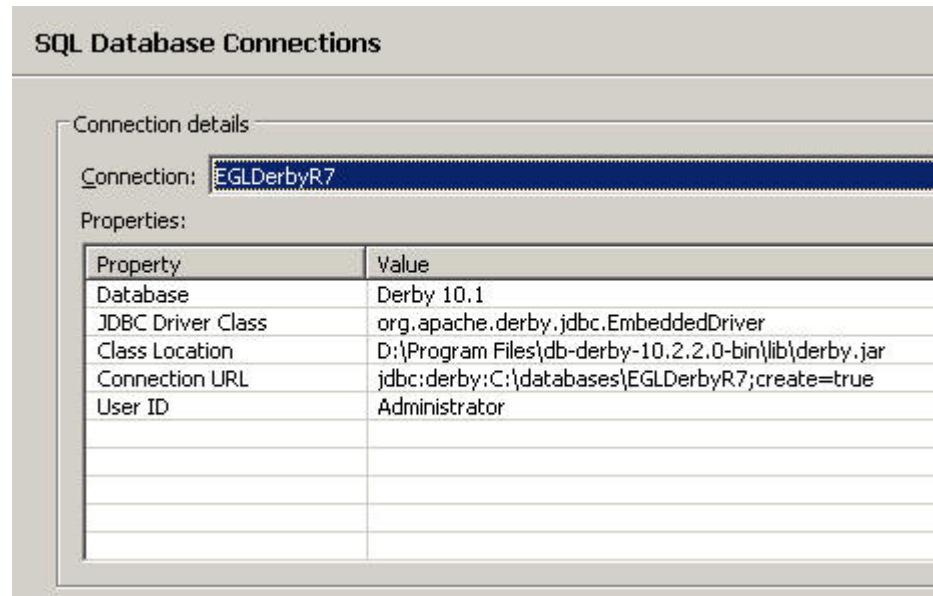
- The basics of working with databases in EGL with statements such as **get** and **add**
- Ways to write your own SQL statements, including in explicit SQL, in the SQL builder, in the default selection condition for SQLRecord parts, in prepared statements, and with the **execute** statement
- How to work with more complex data by performing table joins
- Host variables in explicit SQL and in prepared statements
- How to handle errors in data access applications

Troubleshooting

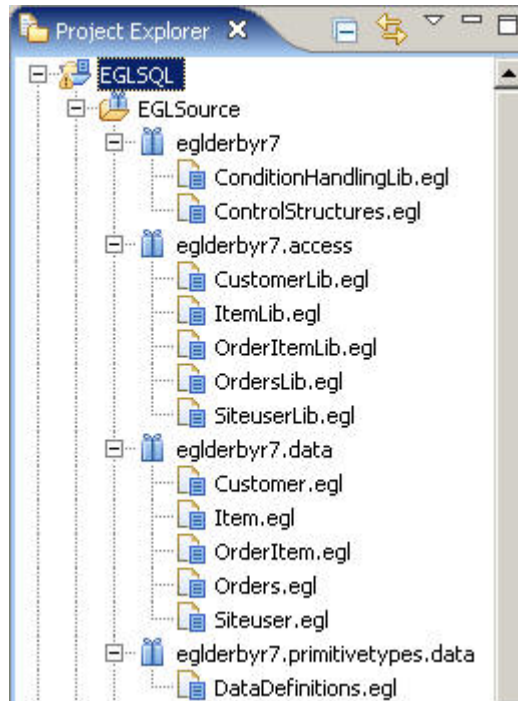
This section lists possible problems in working with the tutorial application. If you are not getting the results that you expect, here are some things you can try to fix the problem.

- Make sure that your EGL code has no validation or generation errors, marked by red X icons in the code editor or by errors in the EGL Generation Results view. If a source file is listed in the EGL Generation Results view with a red X icon rather than a green check mark icon, it did not generate properly. Most of the EGL code used in this tutorial is included at the end for your reference; see “Resources” on page 53.
- Make sure all of your files are saved and then generate the entire project by right-clicking the project and then clicking **Generate**.
- Make sure that you have followed all of the steps in “Lesson 1: Set up the database connection” on page 2, including:
 - Set all the build descriptor options by opening the build descriptor for the project and selecting your database connection in the **Load DB options using Connection** list. Save and generate the project after making any changes to the build descriptor.
 - Add the Derby JAR file to the Java build path for the project.
- Check that the database connection is correct:

1. Click **Window** → **Preferences**.
2. In the Preferences window, expand **EGL** and click **SQL Database Connections**.
3. In the **Connection** list, select your connection, which is usually named EGLDerbyR7, unless you have already created a connection to the sample database (such as in the tutorial *Introducing EGL (a quick-start guide)*), in which case the connection is named EGLDerbyR71.
4. Make sure that the properties for that connection match the locations of the relevant files on your computer. Your connection should look like this, with your own locations for the derby.jar file and the EGLDerbyR7 folder (which contains the database):



5. If the properties don't match the actual locations of the files, click **Edit**, correct the settings, and click **Test Connection** to ensure that the connection works.
- Check that the data parts and logic parts have been created correctly:
 - Make sure that you have all of the files in the eglderbyr7 package, as in this picture:



- As an example, open the Customer.egl file in the eglderbyr7.data package and compare the Customer Record part to the following record:

```
record Customer type sqlRecord {
    tablenames=[["EGL.CUSTOMER"]],
    keyItems=[CustomerId]
}

CustomerId CustomerId {column="EGL.CUSTOMER.CUSTOMER_ID";
FirstName FirstName {column="EGL.CUSTOMER.FIRST_NAME",
    sqlVariableLen=yes, maxLen=30, isSqlNullable=yes};
LastName LastName {column="EGL.CUSTOMER.LAST_NAME",
    sqlVariableLen=yes, maxLen=30, isSqlNullable=yes};
Password Password {column="EGL.CUSTOMER.PASSWORD",
    maxLen=8, isSqlNullable=yes};
Phone Phone {column="EGL.CUSTOMER.PHONE",
    sqlVariableLen=yes, maxLen=14, isSqlNullable=yes};
EmailAddress EmailAddress {column="EGL.CUSTOMER.EMAIL_ADDRESS",
    sqlVariableLen=yes, maxLen=50, isSqlNullable=yes};
Street Street {column="EGL.CUSTOMER.STREET",
    sqlVariableLen=yes, maxLen=30, isSqlNullable=yes};
Apartment Apartment {column="EGL.CUSTOMER.APARTMENT",
    sqlVariableLen=yes, maxLen=10, isSqlNullable=yes};
City City {column="EGL.CUSTOMER.CITY", sqlVariableLen=yes,
    maxLen=30, isSqlNullable=yes};
State State {column="EGL.CUSTOMER.\"STATE\"", maxLen=2,
    isSqlNullable=yes};
Postalcode Postalcode {column="EGL.CUSTOMER.POSTALCODE",
    sqlVariableLen=yes, maxLen=10, isSqlNullable=yes};
Directions Directions {column="EGL.CUSTOMER.DIRECTIONS",
    sqlVariableLen=yes, maxLen=255, isSqlNullable=yes};
end
```

If your parts aren't correct, step through the EGL Data Access Application wizard again carefully to recreate them and overwrite the incorrect ones. For example, if the **tableNames** property on your Record parts and the **column** property on the fields within those Record parts do not include the EGL. prefix to each table name, you forgot to select the **Qualify table names with schema** check box.

Runtime error messages

This is a list of some error messages that you may see in the Console view while trying to run the programs in this tutorial.

EGL0505E Cannot connect to jdbc:derby:C:\databases\EGLDerbyR7;create=true: Failed to start database 'C:\databases\EGLDerbyR7', see the next exception for details.

The data tools that create parts based on the database are still connected to the database, preventing the EGL program from connecting. Close the data tools connection in the Database Explorer view and then try running the EGL program again, as explained in “Running a test program” on page 8.

EGL0507E An error occurred while loading the JDBC drivers. Error: egl.core.RuntimeException: EGL0009E A value for the vgj.jdbc.drivers property is required.

The build descriptor is not set correctly. Load values for the build descriptor options by opening the build descriptor and selecting your connection in the **Load DB options using Connection** list.

EGL0507E An error occurred while loading the JDBC drivers. Error: java.lang.ClassNotFoundException: org.apache.derby.jdbc.EmbeddedDriver

EGL cannot find the derby.jar file. Make sure you have added the file to the project's Java build path as explained in “Importing and connecting to the database” on page 3.

Resources

This tutorial used the following resources:

- A sample Derby database, found at the following location:

shared_resources/plugins/com.ibm.etools.egl.tutorial10001.doc_version/resources/EGLDerbyR7.zip

shared_resources

The shared resources directory for your product, such as C:\Program Files\IBM\SDP70Shared on a Windows system or /opt/IBM/SDP70Shared on a Linux system. If you installed and kept a previous version of an IBM product containing EGL before installing your current product, you may need to specify the shared resources directory that was set up in the earlier install.

version

The installed version of the plugin, including three numbers separated by periods, a string separator, and the date and time that the plugin was built; for example, 7.0.0.RFB_20070120_1300. If more than one is present, use the one with the most recent version number, unless you have a reason to use an older version.

- “Completed CRUDTest.egl file after lesson 2” on page 54
- “Completed selectItems.egl file after lesson 3” on page 56
- “Completed selectItems.egl file after lesson 4A” on page 56
- “Completed selectItems.egl file after lesson 4B” on page 58
- “Completed printCustomerOrders.egl and records.egl files after lesson 5” on page 60
- “Completed CustomerTableOperations.egl and duplicateTable.egl files after lesson 6” on page 62
- “Completed exceptionHandlingTest.egl file after lesson 7” on page 63

Following are some links to the help system on topics covered in this tutorial:

- #sql directive
- Working with SQL data
- SQLRecord stereotype
- add considerations for SQL
- delete considerations for SQL
- execute considerations for SQL
- forEach considerations for SQL
- get considerations for SQL
- open considerations for SQL
- prepare considerations for SQL
- replace considerations for SQL
- EGL Java runtime error codes
- EGL core exception records
- EGL library sqlLib
- commit()
- rollback()
- Working with SQL statements

Completed CRUDTest.egl file after lesson 2

This code is the completed version of the CRUDTest.egl file. If you see any errors marked by red X symbols in the file, make sure your code matches this code:

```
package programs;

import egl Derby7.data.*;

program CRUDTest type BasicProgram {}

    function main()

        // Print the starting records in the database.
        SysLib.writeStdout("Contents of database before any SQL operations:");
        printAllCustomers();

        // Add a record.
        addCustomer();

        // Update the record.
        updateCustomer();

        // Delete a record.
        deleteCustomer();

    end

    function addCustomer()
        customerToAdd Customer;
        customerToAdd.CustomerId = 50;
        customerToAdd.FirstName = "John";
        customerToAdd.LastName = "Doe";

        // Insert the record into the database
        try
            add customerToAdd;
            SysLib.writeStdout("Contents of database after adding a new record:");
            printAllCustomers();
        onException(exception SQLException)
            handleDBException(exception);
    end
end
```

```

        end

    end

function deleteCustomer()
    customerToDelete Customer;
    customerToDelete.CustomerId = 50;

    // Delete the record.
    try
        delete customerToDelete noCursor;
        SysLib.writeStdout("Contents of database after deleting the record:");
        printAllCustomers();
    onException(exception SQLException)
        handleDBException(exception);
    end

end

function updateCustomer()
    customerToUpdate Customer;
    customerToUpdate.CustomerId = 50;

    // Retrieve the row.
    get customerToUpdate;

    // Change values in the record variable.
    customerToUpdate.FirstName = "Johnny";
    customerToUpdate.LastName = "Five";

    // Update the row in the database.
    try
        replace customerToUpdate noCursor;
        SysLib.writeStdout("Contents of database after updating the record:");
        printAllCustomers();
    onException(exception SQLException)
        handleDBException(exception);
    end

end

// This function prints out the contents of the CUSTOMER
// table in the database.
function printAllCustomers()

    allCustomers Customer[0];

    get allCustomers;
    for (counter int from 1 to allCustomers.getSize())
        SysLib.writeStdout(allCustomers[counter].CustomerId::" "::
            allCustomers[counter].FirstName::" "::
            allCustomers[counter].LastName);
    end
    SysLib.writeStdout("\n");

end

//This function responds to errors accessing the database.
function handleDBException(exception SQLException)
    SysLib.writeStdout("Error accessing database. "::
        "See Troubleshooting section");
    SysLib.writeStdout(exception.message);
end

end

```

Return to “Lesson 2: Basic SQL operations in EGL” on page 9.

Completed selectItems.egl file after lesson 3

This code is the completed version of the selectItems.egl file after lesson 3. If you see any errors marked by red X symbols in the file, make sure your code matches this code:

```
package programs;

import egl Derby7.data.Item;
import egl Derby7.primitiveTypes.data.Price;

program selectItems type BasicProgram {}

function main()
  try
    printExpensiveItems(200);
  onException(exception SQLException)
    handleDBException(exception);
  end
end

function printExpensiveItems(maxPrice Price in)

  tempItem Item;
  open expensiveItems for tempItem with
    #sql{
      select
        EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
        EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
      from EGL.ITEM
      where
        EGL.ITEM.PRICE >= :maxPrice
      order by
        EGL.ITEM.ITEM_ID asc
    };

  forEach (tempItem)
    SysLib.writeStdout(tempItem.Name :: " " ::
      tempItem.Description :: " $" :: tempItem.Price);
  end

end

function handleDBException(exception SQLException)
  SysLib.writeStdout("Error accessing database. "::
    "See Troubleshooting section");
  SysLib.writeStdout(exception.message);
end

end
```

Return to “Lesson 3: Managing a cursor with **open** and **forEach**” on page 17.

Completed selectItems.egl file after lesson 4A

This code is the completed version of the selectItems.egl file after lesson 4A. If you see any errors marked by red X symbols in the file, make sure your code matches this code:

```
package programs;

import egl Derby7.data.Item;
import egl Derby7.primitiveTypes.data.Price;

program selectItems type BasicProgram {}

function main()
```

```

minPrice, maxPrice Price?;
minPrice = 50;
maxPrice = 500;
try
    printItemRange(minPrice, maxPrice);
onException(exception SQLException)
    handledDBException(exception);
end
end

function printItemRange(minPrice Price? in, maxPrice Price? in)

    selectStatement STRING = "select * from EGL.ITEM where ";

    if (minPrice == NULL && maxPrice == NULL)
        // Two empty parameters;
        // return all rows with PRICE greater than zero.
        selectStatement ::= "EGL.ITEM.PRICE > 0 ";
    else
        if (minPrice != NULL && maxPrice != NULL)
            // Both parameters were specified.
            selectStatement ::= "EGL.ITEM.PRICE between ":: minPrice :: " and " :: maxPrice :: " ";
        else
            // Only one parameter was specified.
            if (minPrice == NULL)
                // Maximum was specified.
                selectStatement ::= "EGL.ITEM.PRICE <= " :: maxPrice :: " ";
            else
                // Minimum was specified.
                selectStatement ::= "EGL.ITEM.PRICE >= ":: minPrice :: " ";
            end
        end
    end

end

// Regardless of parameters, sort by price.
selectStatement ::= "order by EGL.ITEM.PRICE";

// Print completed SQL code.
SysLib.writeStdout("Completed query: "::selectStatement);

// Prepare the statement to be used.
tempItem Item;
prepare preparedStatement
    from selectStatement
    for tempItem;

// Use the prepared statement in place of explicit SQL.
open expensiveItems for tempItem with preparedStatement;

// Print the results to the console.
forEach (tempItem)
    SysLib.writeStdout(tempItem.Name :: " " ::
        tempItem.Description :: " $" :: tempItem.Price);
end

end

function printExpensiveItems(maxPrice Price in)

tempItem Item;
open expensiveItems for tempItem with
    #sql{
        select
            EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
            EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
    }

```

```

        from EGL.ITEM
        where
            EGL.ITEM.PRICE >= :maxPrice
        order by
            EGL.ITEM.ITEM_ID asc
    };

    foreach (tempItem)
        SysLib.writeStdout(tempItem.Name :: " " ::
            tempItem.Description :: " $" :: tempItem.Price);
    end

end

function handleDBException(exception SQLException)
    SysLib.writeStdout("Error accessing database. "::
        "See Troubleshooting section");
    SysLib.writeStdout(exception.message);
end

end

```

Return to “Lesson 4A: Using a prepared statement” on page 24.

Completed selectItems.egl file after lesson 4B

This code is the completed version of the selectItems.egl file after lesson 4B. If you see any errors marked by red X symbols in the file, make sure your code matches this code:

```

package programs;

import eglderbyr7.data.Item;
import eglderbyr7.primitivetypes.data.Price;

program selectItems type BasicProgram {}

function main()
    minPrice, maxPrice Price?;
    minPrice = 50;
    maxPrice = 500;
    try
        printItemRange2(minPrice, maxPrice);
    onException(exception SQLException)
        handleDBException(exception);
    end
end

function printItemRange2(minPrice Price? in, maxPrice Price? in)

    selectStatement STRING = "select * from EGL.ITEM where " ::
        "EGL.ITEM.PRICE between ? and ? " ::
        "order by EGL.ITEM.PRICE";

    // Prepare the statement to be used.
    tempItem Item;
    prepare preparedStatement
        from selectStatement
        for tempItem;

    mostExpensiveItem Item;
    get mostExpensiveItem
        into mostExpensiveItem.Price
        with
        #sql{
            select
                max(EGL.ITEM.PRICE)

```

```

        from EGL.ITEM
    };

    if (minPrice == NULL && maxPrice == NULL)
        // Two empty parameters;
        open expensiveItems for tempItem with preparedStatement
            using 0, mostExpensiveItem.Price;
    else

        if (minPrice != NULL && maxPrice != NULL)
            // Both parameters were specified.
            open expensiveItems for tempItem with preparedStatement
                using minPrice, maxPrice;
        else
            // Only one parameter was specified.
            if (minPrice == NULL)
                // Maximum was specified.
                open expensiveItems for tempItem with preparedStatement
                    using 0, maxPrice;
            else
                // Minimum was specified.
                open expensiveItems for tempItem with preparedStatement
                    using minPrice, mostExpensiveItem.Price;
            end
        end
    end

    // Print the results to the console.
    foreach (tempItem)
        SysLib.writeStdout(tempItem.Name :: " " ::
            tempItem.Description :: " $" :: tempItem.Price);
    end

end

function printItemRange(minPrice Price? in, maxPrice Price? in)

    selectStatement STRING = "select * from EGL.ITEM where ";

    if (minPrice == NULL && maxPrice == NULL)
        // Two empty parameters;
        // return all rows with PRICE greater than zero.
        SysLib.writeStdout("All items with price greater than zero:");
        selectStatement ::= "EGL.ITEM.PRICE > 0 ";
    else

        if (minPrice != NULL && maxPrice != NULL)
            // Both parameters were specified.
            selectStatement ::= "EGL.ITEM.PRICE between ":: minPrice :: " and " :: maxPrice :: " ";
        else
            // Only one parameter was specified.
            if (minPrice == NULL)
                // Maximum was specified.
                selectStatement ::= "EGL.ITEM.PRICE <= " :: maxPrice :: " ";
            else
                // Minimum was specified.
                selectStatement ::= "EGL.ITEM.PRICE >= ":: minPrice :: " ";
            end
        end
    end

    // Regardless of parameters, sort by price.
    selectStatement ::= "order by EGL.ITEM.PRICE";

    // Print completed SQL code.

```

```

SysLib.writeStdout("Completed query: "::selectStatement);

// Prepare the statement to be used.
tempItem Item;
prepare preparedStatement
  from selectStatement
  for tempItem;

// Use the prepared statement in place of explicit SQL.
open expensiveItems for tempItem with preparedStatement;

// Print the results to the console.
forEach (tempItem)
  SysLib.writeStdout(tempItem.Name :: " " ::
    tempItem.Description :: " $" :: tempItem.Price);
end

end

function printExpensiveItems(maxPrice Price in)

tempItem Item;
open expensiveItems for tempItem with
  #sql{
    select
      EGL.ITEM.ITEM_ID, EGL.ITEM.NAME, EGL.ITEM.IMAGE,
      EGL.ITEM.PRICE, EGL.ITEM.DESCRPTION
    from EGL.ITEM
    where
      EGL.ITEM.PRICE >= :maxPrice
    order by
      EGL.ITEM.ITEM_ID asc
  };

forEach (tempItem)
  SysLib.writeStdout(tempItem.Name :: " " ::
    tempItem.Description :: " $" :: tempItem.Price);
end

end

function handleDBException(exception SQLException)
  SysLib.writeStdout("Error accessing database. "::
    "See Troubleshooting section");
  SysLib.writeStdout(exception.message);
end

end

```

Return to “Lesson 4B: Host variables in **prepare** statements” on page 27.

Completed printCustomerOrders.egl and records.egl files after lesson 5

This code is the completed version of the printCustomerOrders.egl and records.egl files after lesson 5. If you see any errors marked by red X symbols in either file, make sure your code matches this code:

[data/records.egl]

```

package data;

record OrderCustomerJoin type SQLRecord
  {tableNames = [{"EGL.CUSTOMER", "C"}, [{"EGL.ORDERS", "O"}]},
  keyItems=[CUSTOMER_ID, ORDER_ID],
  defaultSelectCondition = #sqlCondition{ O.CUSTOMER_ID = C.CUSTOMER_ID }}

```

```

CUSTOMER_ID int
{column="C.CUSTOMER_ID", isReadOnly=yes};
FIRST_NAME string
{column="C.FIRST_NAME", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=30};
LAST_NAME string
{column="C.LAST_NAME", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=30};
PASSWORD string
{column="C.PASSWORD", isReadOnly=yes,
 isSqlNullable=yes, maxLen=8};
PHONE string
{column="C.PHONE", isReadOnly=yes, isSqlNullable=yes,
 sqlVariableLen=yes, maxLen=14};
EMAIL_ADDRESS string
{column="C.EMAIL_ADDRESS", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=50};
STREET string
{column="C.STREET", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=30};
APARTMENT string
{column="C.APARTMENT", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=10};
CITY string
{column="C.CITY", isReadOnly=yes, isSqlNullable=yes,
 sqlVariableLen=yes, maxLen=30};
STATE string
{column="C.STATE", isReadOnly=yes,
 isSqlNullable=yes, maxLen=2};
POSTALCODE string
{column="C.POSTALCODE", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=10};
DIRECTIONS string
{column="C.DIRECTIONS", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=255};
ORDER_ID int
{column="O.ORDER_ID", isReadOnly=yes};
orders_CUSTOMER_ID int
{column="O.CUSTOMER_ID", isReadOnly=yes,
 isSqlNullable=yes};
ORDER_AMOUNT decimal(8,2)
{column="O.ORDER_AMOUNT", isReadOnly=yes,
 isSqlNullable=yes};
ORDER_DETAILS string
{column="O.ORDER_DETAILS", isReadOnly=yes,
 isSqlNullable=yes, sqlVariableLen=yes, maxLen=111};
ORDER_DATE date
{column="O.ORDER_DATE", isReadOnly=yes,
 isSqlNullable=yes};
ORDER_STATUS string
{column="O.ORDER_STATUS", isReadOnly=yes,
 isSqlNullable=yes, maxLen=12};
end

[programs/printCustomerOrders.egl]

package programs;

import data.OrderCustomerJoin;

program printCustomerOrders type BasicProgram {}

function main()

    allCustomerOrders OrderCustomerJoin[0];

```

```

tempRec OrderCustomerJoin;
open resultSet for tempRec;
foreach (tempRec)
  SysLib.writeStdout(tempRec.CUSTOMER_ID :: " "
    :: tempRec.LAST_NAME :: " " :: tempRec.ORDER_ID);
end

end

end

```

Return to “Lesson 5: Retrieving more complex data with table joins” on page 31.

Completed CustomerTableOperations.egl and duplicateTable.egl files after lesson 6

This code is the completed version of the CustomerTableOperations.egl and duplicateTable.egl files after lesson 6. If you see any errors marked by red X symbols in either file, make sure your code matches this code:

```

[libraries/CustomerTableOperations.egl]

package libraries;

import egl Derby7.ConditionHandlingLib;
import egl Derby7.StatusRec;

library CustomerTableOperations type BasicLibrary {}

function execCreateTable(status StatusRec inOut)
  try
    execute
      #sql{
        CREATE TABLE EGL.CUSTOMERTEMP (
          CUSTOMER_ID INTEGER NOT NULL,
          FIRST_NAME VARCHAR(30),
          LAST_NAME VARCHAR(30),
          PASSWORD CHAR(8),
          PHONE VARCHAR(14),
          EMAIL_ADDRESS VARCHAR(50),
          STREET VARCHAR(30),
          APARTMENT VARCHAR(10),
          CITY VARCHAR(30),
          STATE CHAR(2),
          POSTALCODE VARCHAR(10),
          DIRECTIONS VARCHAR(255)
        )
      };
    onException (exception SQLException)
      ConditionHandlingLib.HandleException(status, exception);
  end

end

function execInsertIntoTable(status StatusRec inOut)
  try
    execute #sql{
      INSERT INTO EGL.CUSTOMERTEMP
      SELECT * FROM EGL.CUSTOMER
    };
    onException (exception SQLException)
      ConditionHandlingLib.HandleException(status, exception);
  end

end

function execDropTable(status StatusRec inOut)

```

```

        try
            execute #sql{
                DROP TABLE EGL.CUSTOMERTEMP
            };
        onException (exception SQLException)
            ConditionHandlingLib.HandleException(status, exception);
        end
    end
end

```

[programs/duplicateTable.egl]

```

package programs;

import egllderbyr7.StatusRec;
import libraries.CustomerTableOperations;

program duplicateTable type BasicProgram {}

    status StatusRec;

    function main()
        CustomerTableOperations.execCreateTable(status);
        CustomerTableOperations.execInsertIntoTable(status);
        //CustomerTableOperations.execDropTable(status);
    end
end

```

Return to “Lesson 6: Executing arbitrary SQL code” on page 35.

Completed exceptionHandlingTest.egl file after lesson 7

This code is the completed version of the exceptionHandlingTest.egl file after lesson 7. If you see any errors marked by red X symbols in the file, make sure your code matches this code:

```

package programs;

import egllderbyr7.data.Customer;
import egllderbyr7.data.Orders;

program exceptionHandlingTest type BasicProgram {}

    function main()
        customer Customer;
        customer.FirstName = "John";
        customer.LastName = "Doe";
        customer.CustomerId = 1;

        customerOrder Orders;
        customerOrder.CustomerId = 1;
        customerOrder.OrderId = 50;
        customerOrder.OrderAmount = 0;
        customerOrder.OrderDetails =
            "This is my first order, so get it right!";

        try
            add customerOrder;
            SysLib.writeStdout("Order added successfully.");
            add customer;
            SysLib.writeStdout("Customer added successfully.");
        onException(ex SQLException)
            SysLib.writeStdout("SQL exception ">::ex.messageID);
            SysLib.writeStdout("message = ">::ex.message);
        end
    end
end

```

```

        SysLib.writeStdout("Performing rollback.");
        SysLib.rollback();
    end

    tempOrder Orders;
    tempOrder.OrderId = 50;
    get tempOrder;

    if (tempOrder is noRecordFound)
        SysLib.writeStdout("The order was rolled back successfully.");
    else
        SysLib.writeStdout("The order is still in the database");
    end

end

end

```

Return to “Lesson 7: Exception handling and rollbacks” on page 45.

Appendix. Notices

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 grant 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.

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:

Intellectual Property Dept. for Rational Software
IBM Corporation
3600 Steeles Avenue East
Markham, ON Canada L3R 9Z7.

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 document 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.

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.

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. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM
Rational

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.



Printed in USA