

Rational Business Developer



チュートリアル: EGL および SQL を使用したリレーショナル・データベースへのアクセス

バージョン 7.1

Rational Business Developer



チュートリアル: EGL および SQL を使用したリレーショナル・データベースへのアクセス

バージョン 7.1

ご注意

本書および本書で紹介する製品をご使用になる前に、67 ページの『特記事項』に記載されている情報をお読みください。

本書は Rational Business Developer のバージョン 7.1 および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： Rational Business Developer
Tutorial: Access relational databases with EGL and SQL
Version 7.1

発行： 日本アイ・ピー・エム株式会社

担当： ナショナル・ランゲージ・サポート

第1刷 2008.5

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

目次

EGL および SQL を使用したリレーショナル・データベースへのアクセス 1

概要	1
演習 1: データベース接続のセットアップ	2
EGL プロジェクトの作成	2
データベースへのインポートおよび接続	4
テスト・プログラムの実行	8
演習 2: EGL での SQL の基本操作	10
CRUD 操作の復習	13
演習のチェックポイント	18
演習 3: open および forEach を使用したカーソルの管理	18
結果セットのステップスルー	21
演習のチェックポイント	23
演習 4: prepare を使用したクエリーの動的構成	23
準備済みステートメントでよく発生するエラー	24
演習 4A: 準備済みステートメントの使用	25
演習 4B: prepare ステートメント内のホスト変数	28
演習のチェックポイント	32
演習 5: テーブル結合を使用した、より複雑なデータの取得	32
演習 6: 任意の SQL コードの実行	36
SQL ビルダーを使用した SQL ステートメントの作成	37

execute の使用	44
演習のチェックポイント	47
演習 7: 例外処理およびロールバック	47
SQL データ・アクセスでの例外処理	47
データベースへの変更のロールバック	49
演習のチェックポイント	51
要約	52
トラブルシューティング	52
実行時のエラー・メッセージ	54
リソース	55
演習 2 で完成した <code>CRUDTest.egl</code> ファイル	56
演習 3 で完成した <code>selectItems.egl</code> ファイル	57
演習 4A で完成した <code>selectItems.egl</code> ファイル	58
演習 4B で完成した <code>selectItems.egl</code> ファイル	60
演習 5 で完成した <code>printCustomerOrders.egl</code> および <code>records.egl</code> ファイル	62
演習 6 で完成した <code>CustomerTableOperations.egl</code> および <code>duplicateTable.egl</code> ファイル	64
演習 7 で完成した <code>exceptionHandlingTest.egl</code> ファイル	65

付録. 特記事項.	67
商標	68

EGL および SQL を使用したリレーショナル・データベースへのアクセス

このチュートリアルでは、ワークベンチで EGL およびその他のツールを使用して、リレーショナル・データベースの作業を行います。EGL によって生成されたデフォルトの SQL コードを使用する方法、および独自の SQL コードを作成して、そのカスタム SQL を EGL アプリケーションと統合する方法を学習します。

学習目標

このチュートリアルでは、中級レベルのユーザーを対象に、EGL を使用してリレーショナル・データベースにアクセスする方法を学習します。次のトピックが含まれています。

- EGL を使用した 4 つの基本的なデータ・アクセス操作 (作成、読み取り、更新、および削除) の実行方法
- EGL が作成した SQL コードをカスタマイズする方法
- EGL コードの内外にカスタム SQL ステートメントを作成し、実行する方法
- データ・アクセス・アプリケーションのエラーを処理する方法

必要な時間

90 分

関連情報

EGL の紹介 (クイック・スタート・ガイド)

EGL での Hello world サービスの作成

EGL での Hello world プログラムの作成

EGL を使った JSF 検索ページの作成



PDF バージョンの表示

概要

このチュートリアルでは、ワークベンチで EGL およびその他のツールを使用して、リレーショナル・データベースの作業を行います。EGL によって生成されたデフォルトの SQL コードを使用する方法、および独自の SQL コードを作成して、そのカスタム SQL を EGL アプリケーションと統合する方法を学習します。

エンタープライズ開発言語 (EGL) は、全機能搭載型のアプリケーションを迅速に作成するために役立つ開発環境およびプログラミング言語です。EGL を利用することにより、ユーザーは、ソフトウェア・テクノロジーではなく、自らが作成するコードによって対処しなければならないビジネス上の問題に集中することができます。

学習目標

このチュートリアルでは、中級レベルのユーザーを対象に、EGL を使用してリレーショナル・データベースにアクセスする方法を学習します。次のトピックが含まれています。

- EGL を使用した 4 つの基本的なデータ・アクセス操作 (作成、読み取り、更新、および削除) の実行方法
- EGL が作成した SQL コードをカスタマイズする方法
- EGL コードの内外にカスタム SQL ステートメントを作成し、実行する方法
- データ・アクセス・アプリケーションのエラーを処理する方法

必要な時間

このチュートリアル の所要時間は、約 90 分です。このチュートリアルに関する他の概念を参照した場合は、この時間内に終わらない可能性がありますのでご注意ください。

スキル・レベル

中級

前提条件

このチュートリアルを開始する前に、SQL およびリレーショナル・データベースに関する中級レベルの知識を身に付けておく必要があります。また、「*EGL の紹介 (クイック・スタート・ガイド)*」などのチュートリアルで EGL の基本を学習した方は、EGL コードの例に沿って作業する方が簡単です。

期待される結果

オプション: コード・スニペット、スクリーン・ショット、マルチメディアのリンク、またはチュートリアル の完了時に期待される結果のテキスト記述を提供できます。つまり、表示、デモ、または説明を提供できるエンド製品について、これまでにそれらの作業を行っていない場合は、ここで行うことができます。

演習 1: データベース接続のセットアップ

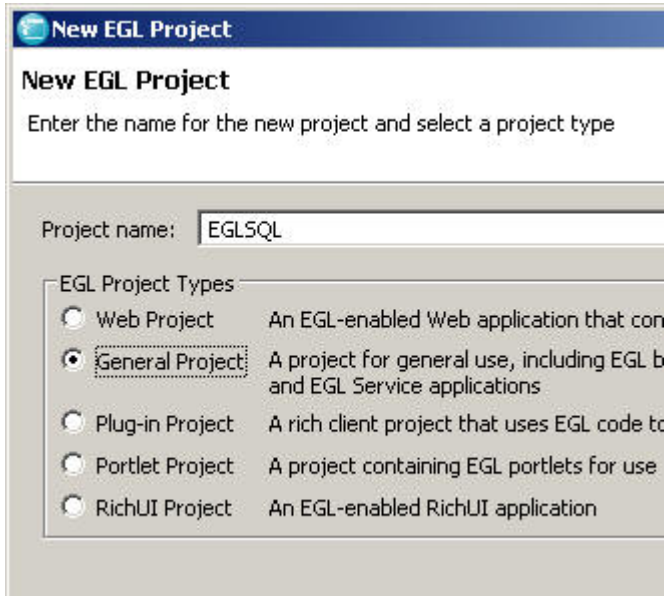
データベースにアクセスするための最初のステップは、接続をセットアップすることです。その接続から、EGL はデータ・パーツおよびロジック・パーツの開始セットを作成し、お客様は単純なデータ・アクセス・プログラムを作成して接続をテストします。

EGL プロジェクトの作成

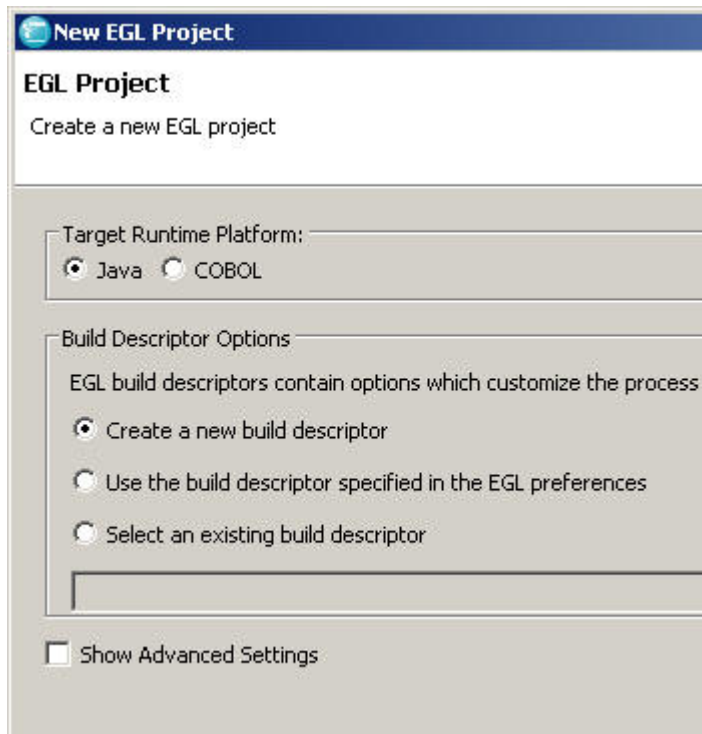
まず、データ・アクセス・コードを保持する新規 EGL プロジェクトが必要です。

1. EGL パースペクティブへの切り替え:
 - a. 「ウィンドウ」 → 「パースペクティブを開く」 → 「その他」の順にクリックします。
 - b. 「パースペクティブを開く」ウィンドウで、「EGL」をクリックします。パースペクティブのリストに EGL が表示されない場合は、「すべて表示」チェック・ボックスを選択します。
 - c. 「OK」をクリックします。
2. 次のようにして、EGLSQL という名前の新規 EGL プロジェクト (EGL Web プロジェクトではありません) を作成します。
 - a. 「ファイル」 → 「新規」 → 「プロジェクト」の順にクリックします。
 - b. 「新規プロジェクト」ウィンドウで、「EGL」を展開して、「EGL プロジェクト・ウィザード」をクリックします。
 - c. 「次へ」をクリックします。
 - d. 「プロジェクト名」フィールドに、EGLSQL と入力します。

- e. 「EGL プロジェクト・タイプ (EGL Project Types)」で、「汎用プロジェクト (General Project)」をクリックします。「新規 EGL プロジェクト」ウィンドウは次のようになります。



- f. 「次へ」をクリックします。
- g. 2 ページ目で、「Java」が「ターゲット・ランタイム・プラットフォーム」で選択され、「新規ビルド記述子の作成 (Create a new build descriptor)」が「ビルド記述子オプション」で選択されていることを確認します。「新規 EGL プロジェクト」ウィンドウは次のようになります。



- h. 「終了」をクリックします。
- 新規プロジェクトが作成され、「プロジェクト・エクスプローラー」ビューに表示されます。

データベースへのインポートおよび接続

このチュートリアルでは、チュートリアル「*EGL の紹介 (クイック・スタート・ガイド)*」で使用するサンプルの Derby データベースを使用します。次のステップで、このデータベースのもう 1 つのコピーをご使用のコンピューター上の任意のロケーションに unzip します。また、「データ・アクセス・アプリケーション」ウィザードを使用してデータベースに接続し、そのデータベースに基づいてデータ・パーツとロジック・パーツを作成します。これらのパーツについて詳しくは、「*EGL の紹介 (クイック・スタート・ガイド)*」を参照してください。

1. 次のリンクをクリックして、コンピューターの一時フォルダー (デスクトップなど) に、サンプル・データベースをダウンロードします。

サンプル・データベース

後で見つけることができるのであれば、データベースはどこに保管してもかまいません。

このサンプル・データベースは、次のロケーションの製品インストール・ディレクトリーにもあります。

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

shared_resources

ご使用の製品用の共用リソース・ディレクトリーで、例えば、Windows システムの場合は `C:\Program Files\IBM\SDP70Shared`、Linux システムの場合は `/opt/IBM/SDP70Shared` です。現行製品をインストールする以前より、EGL を含む IBM 製品の前のバージョンをインストールし、保持している場合は、前のインストールでセットアップされた共用リソース・ディレクトリーを指定する必要があります。

version

インストールされているプラグインのバージョン。これには、ピリオドで区切られた 3 つの数字、ストリング区切り文字、およびプラグインがビルドされた日時が含まれます。例:

`7.0.0.RFB_20070120_1300`。複数存在している場合は、古いバージョンを使用する理由がない限り、最も新しいバージョン番号のプラグインを使用してください。

2. データベースをご使用のコンピューター上の `C:\databases` などのロケーションに unzip します。この場合も、後で再度データベースを見つけることができるのであれば、どこに置いてかまいません。
3. 「ファイル」 → 「新規」 → 「その他」の順にクリックします。「新規」ウィンドウが開きます。
4. 「EGL」を展開して「EGL データ・アクセス・アプリケーション」をクリックします。
5. 「次へ」をクリックします。
6. 「プロジェクト設定の定義」ページの「プロジェクト名」リストから、「EGLSQL」を選択します。
7. 「データベース接続」の横にある「新規」をクリックします。
8. 「新規接続」ウィンドウの「データベース・マネージャーの選択」で、「Derby」を展開して「10.1」をクリックします。
9. 「データベース・ロケーション」フィールドで、先程 unzip した zip ファイル内にある EGLDerbyR7 フォルダーを参照します。例えば、EGLDerbyR7.zip ファイルを `C:\databases` に unzip した場合、「データベース・ロケーション」フィールドは `C:\databases\EGLDerbyR7` となります。

「ユーザー ID」または「パスワード」フィールドは、変更する必要はありません。
10. 「クラス・ロケーション」フィールドに、`derby.jar` ファイルへのパスを入力します。

このファイルを見つける方法はいくつかあります。

- IBM® WebSphere® Application Server バージョン 6.1 をインストールした場合は、サーバーに組み込まれている Derby のバージョンを使用することができます。次のフォルダーで derby.jar を検索してください。

`install_location/runtimes/base_v61/derby/lib`

- 製品とともにデータベース・ツールをインストールした場合は、次の場所でファイルを見つけることができます。

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

`shared_resources`

ご使用の製品用の共用リソース・ディレクトリーで、例えば、Windows システムの場合は C:\Program Files\IBM\SDP70Shared、Linux システムの場合は /opt/IBM/SDP70Shared です。現行製品をインストールする以前より、EGL を含む IBM 製品の前のバージョンをインストールし、保持している場合は、前のインストールでセットアップされた共用リソース・ディレクトリーを指定する必要があります。

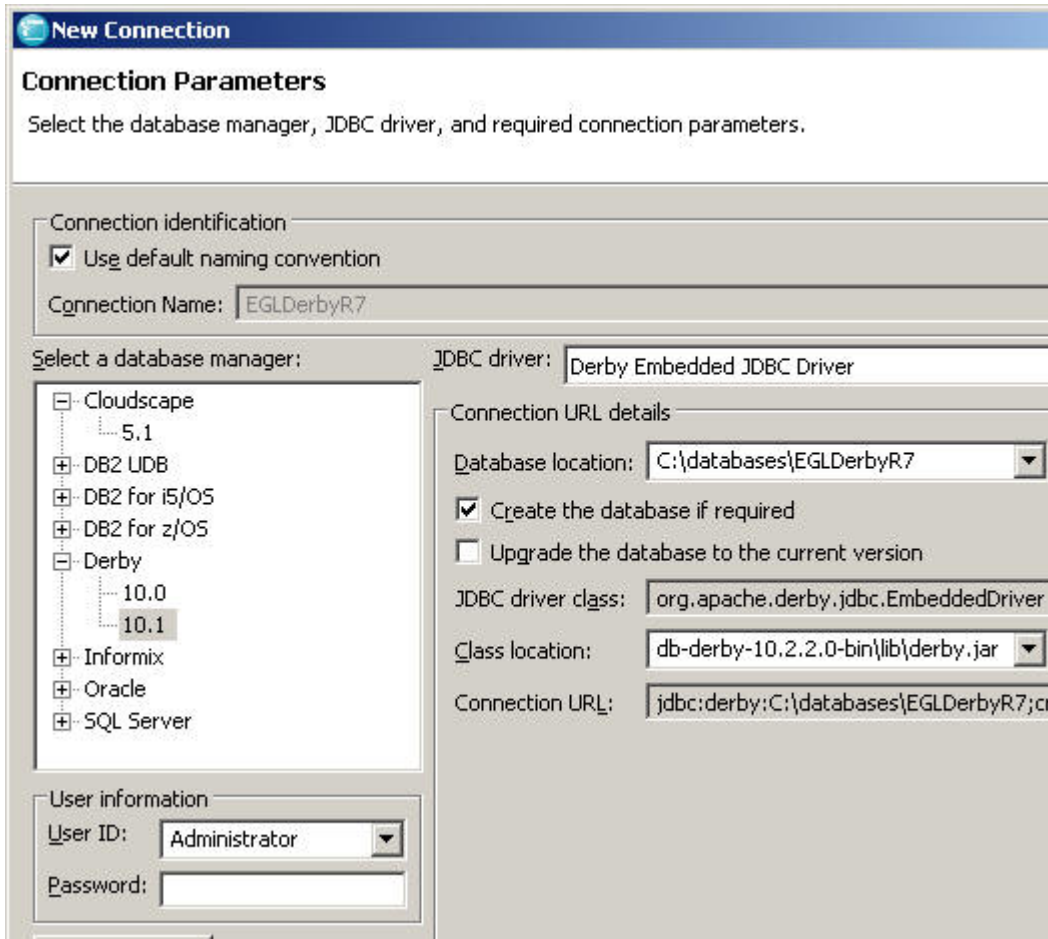
`version`

インストールされているプラグインのバージョン。これには、ピリオドで区切られた 3 つの数字、ストリング区切り文字、およびプラグインがビルドされた日時が含まれます。例:

7.0.0.RFB_20070120_1300。複数存在している場合は、古いバージョンを使用する理由がない限り、最も新しいバージョン番号のプラグインを使用してください。

- 前の 2 つの場所でファイルが見つからない場合は、製品インストール・ディレクトリーで derby.jar ファイルを検索してください。インストール済み環境によって、ファイルの場所は異なる可能性があります。
- ご使用のコンピューター上にファイルが見つからない場合は、Derby Web サイト (<http://db.apache.org/derby/>) から直接ファイルをダウンロードすることができます。最後にリリースされたバージョンの Derby をダウンロードして、ご使用のコンピューター上の場所に derby.jar ファイルを解凍する必要があります。

「新規接続」ウィンドウは、次のようになります (ユーザーのワークスペースおよびロケーション情報は、「データベース・ロケーション」 および「クラス・ロケーション」 フィールドにあります)。



11. 「デフォルトの命名規則を使用 (Use default naming convention)」チェック・ボックスが選択されていることを確認します。
12. 「終了」をクリックします。これで、データベースへの接続が確立されました。データベースのテーブルはすべて、ウィザード下部にある「テーブル名」にリストされています。一部のテーブルにはメタデータしか含まれていないため、すべてのテーブルに対して、データ・パーツを作成することはできません。
13. 「表の選択」において、次のテーブルの横にあるチェック・ボックスのみを選択します。
 - EGL.CUSTOMER
 - EGL.ITEM
 - EGL.ORDERS
 - EGL.ORDER_ITEM
 - EGL.SITEUSER
 - EGL.STATETABLE

「EGL データ・アクセス・アプリケーション」ウィザードは、次のようになります。

EGL Data Access Application

Define project settings

Specify project location and select tables from relational database.

Project Name:

Project contents

☒ Use default location

Project location:

Database Connection:

Select Tables:

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

☐ Create web pages

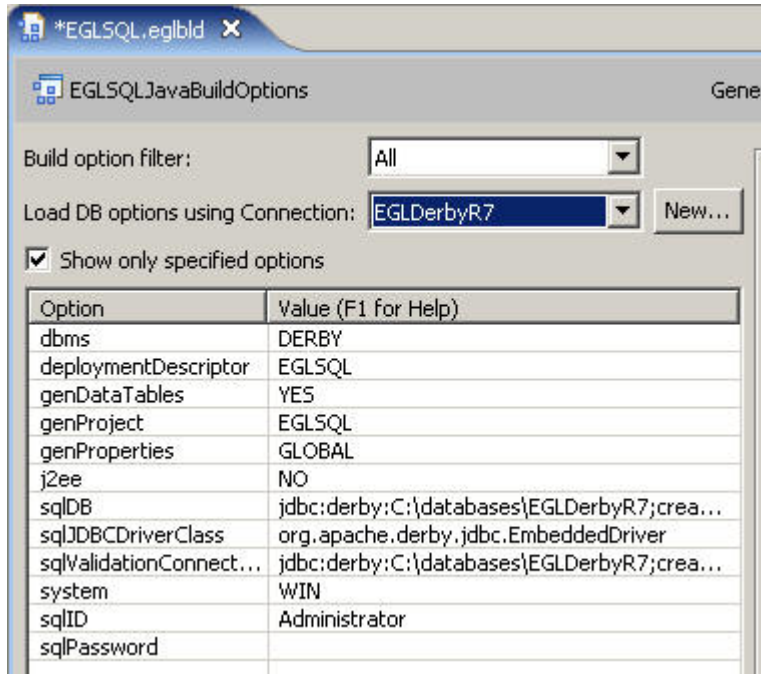
14. 「データベース接続」フィールドにある接続名をメモします。後のステップで、この接続名が必要になります。通常、接続名はデータベース名の後に **EGLDerbyR7** が付いたものです。ただし、このワークスペースでチュートリアル「*EGL の紹介 (クイック・スタート・ガイド)*」を完了している場合は、この名前のデータベース接続が既に作成されているため、EGL により新しい接続名 **EGLDerbyR71** が付けられます。
15. 「Web ページの作成」チェック・ボックスをクリアします。
16. **EGLWeb** プロジェクトが、「プロジェクト名」フィールドにリストされていることを確認してください。

注: まだ「終了」をクリックしないでください! このウィザードで 1 つ以上の設定を変更する必要があります。

17. 「次へ」をクリックして、「フィールドの定義」ページに移動します。このページでは、データベース・テーブルにキー・フィールドを追加することができます。このページでは設定を変更しないでください。データベースの各テーブルには既にキー・フィールドがあります。
18. 再度「次へ」をクリックして、「プロジェクト作成オプションの定義」ページに移動します。
19. 「プロジェクト作成オプションの定義」ページで、「テーブル名をスキーマで修飾」チェック・ボックスを選択します。
20. 「終了」をクリックします。EGL により、データベースにアクセスするために使用可能なさまざまなデータ・パーツおよびロジック・パーツが作成されます。これらのパーツは、この後のチュートリアルで頻繁に使用されます。

このデータベース接続を実行時に使用するために、さらにビルド記述子を構成する必要があります。

21. EGLSQL/EGLSource/EGLSQL.eglbld にあるプロジェクトのデフォルトのビルド記述子を開きます。このファイルをダブルクリックして、EGL ビルド・パーツ・エディターで開きます。
22. EGL ビルド・パーツ・エディターで、「**接続を使用した DB オプションのロード (Load DB options using Connection)**」の横にある、前のステップでメモしたデータベース接続を選択します。ビルド記述子オプションは、その接続の使用に必要な値に設定されています。

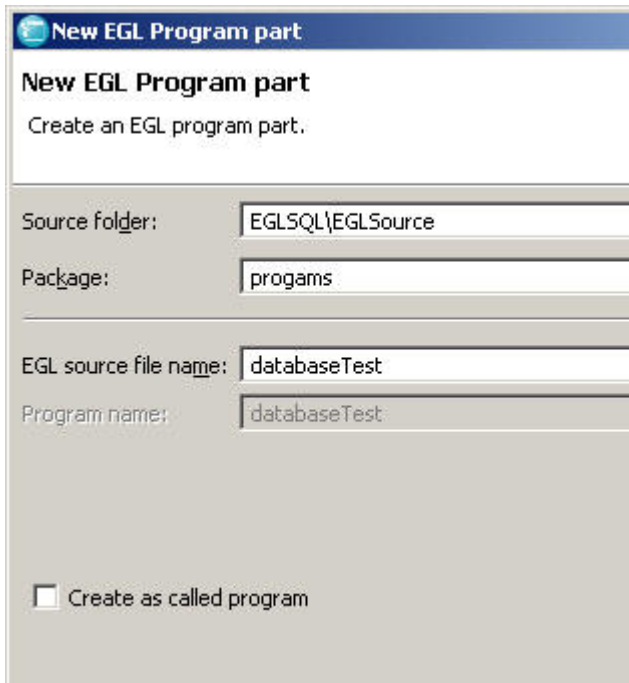


23. ビルド記述子を保管して、閉じます。
24. 次のようにして、Derby ドライバーをプロジェクトの Java™ ビルド・パスに追加します。
 - a. EGLSQL プロジェクトを右クリックし、「プロパティ」をクリックします。
 - b. 「プロパティ」ウィンドウで、「Java のビルド・パス」をクリックします。
 - c. 「Java のビルド・パス」ページで、「ライブラリー」タブを開きます。
 - d. 「ライブラリー」タブで、「外部 JAR の追加」をクリックします。
 - e. 「JAR の選択」ウィンドウで、Derby.jar ファイルを選択し、「開く」をクリックします。
 - f. 「終了」をクリックします。
 - g. 「OK」をクリックします。

テスト・プログラムの実行

データベース接続が機能していることを確認するには、次のステップに従ってテスト・プログラムを実行し、データベースの行をコンソールに出力します。

1. 「ファイル」 → 「新規」 → 「プログラム」の順にクリックします。
2. 「新規 EGL プログラム・パーツ」ウィンドウで、「パッケージ」フィールドに programs と入力します。
3. 「EGL ソース・ファイル名」フィールドに、databaseTest と入力します。「新規 EGL プログラム・パーツ」ウィンドウは次のようになります。



4. 「終了」をクリックします。新規プログラムが作成されてエディターで開きます。
5. 新規プログラム内のすべてのコードを、次のコードで置換します。

```
package programs;

import eglderbyr7.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. プログラムを保存します。
7. 「プロジェクト・エクスプローラー」ビューで EGLSQL を右クリックし、「生成」をクリックして、プロジェクト全体を生成します。

プログラムを実行するには、事前にデータベースからデータベース・ツールを切断しておく必要があります。Derby では一度に 1 つの接続しか許可されません。また、EGL がデータ・パーツの作成およびビルド記述子の値の設定に使用したツールは、接続を閉じるまで接続されたままになります。

8. データベース接続を閉じます。これで、データベースが実行時にプログラムで使用できるようになります。

- a. 「ウィンドウ」 → 「ビューの表示」 → 「その他」の順にクリックし、ビューのリストから「データ」 → 「データベース・エクスプローラー」を選択してから「OK」をクリックして、「データベース・エクスプローラー」ビューを開きます。
 - b. 「データベース・エクスプローラー」ビューで、「接続」を展開します。 EGLDerbyR7 など、「EGL ビルド記述子」で選択したのと同じ名前のデータベース接続が「接続」の下にリストされます。
 - c. データベース接続を右クリックし、「切断」をクリックします。
 - d. 「データベース・エクスプローラー」ビューを閉じます。 このビューなどのデータ・ツールについては、後のチュートリアルでより詳しく学習します。
9. 次のようにして、テスト・プログラムを実行します。
- a. EGLSQL/JavaSource/programs を展開します。 JavaSource フォルダーには、ご使用のプログラムの生成された出力が保持されています。
 - b. databaseTest.java ファイルを右クリックし、「実行」 → 「Java アプリケーション」の順にクリックします。

テスト・プログラムを実行すると、データベースの CUSTOMER テーブルの内容が「コンソール」ビューに出力されます。



```
<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
```

「コンソール」ビューに出力ではなくエラー・メッセージが表示された場合は、52 ページの『トラブルシューティング』を参照してください。

演習 2: EGL での SQL の基本操作

チュートリアル「EGL の紹介 (クイック・スタート・ガイド)」を完了している場合は、リレーショナル・データベースへのアクセスに必要な EGL パーツ、およびデータベースでのデータの読み取りおよび書き込みに使用するコマンドについて既に学習しています。この演習では、これらの概念をより詳細に説明します。

前回の演習または前のチュートリアル of いずれかでデータ・アクセス・アプリケーション・ウィザードを使用した際に、EGL により、データベース内の情報に基づいてデータ・パーツおよびロジック・パーツが作成されました。

作成された中で最も代表的なデータ・パーツは、SQLRecord パーツです。これらのレコード・パーツ内のフィールドは、データベース内の列に関連しています。このレコードに基づいて変数を作成すると、その変数を使用してデータベース内の新規行または既存行を示すことができます。データ・アクセス・アプリケーション・ウィザードで作成されたそれぞれの SQLRecord は単一テーブル内の列とのみ一致しますが、この後のチュートリアルでは、複数のテーブルから単一レコードにデータをマージする方法 (SQL では「テーブル結合」といいます) を学習します。

例えば、このチュートリアル全体を通して Customer SQLRecord を使用します。このレコードは、ファイル EGLSource/eglderbyr7/data/customer.egl にあります。

```
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
```

コード `tablenames=[["EGL.CUSTOMER"]]` はこのレコードと関連するデータベース内のテーブルを指定し、各フィールドの **column** プロパティはフィールドと関連するテーブル内の列を指定します。コード `keyItems=[CustomerId]` は、CustomerId フィールド (および接続により、データベースの EGL.CUSTOMER.CUSTOMER_ID 列) が、このテーブルの主キーであることを示しています。

データ・アクセス・アプリケーション・ウィザードにより、いくつかのライブラリーも作成されました。このライブラリーは、レコードおよびレコードの配列に対し、CRUD (作成、読み取り、更新、および削除) 操作を実行します。これらのライブラリーは、EGL コマンドを使用して直接 CRUD 操作を実行するための、より洗練された方法です。このチュートリアルでは EGL コマンドを直接使用しますが、もちろん、ロジックをライブラリーに分割して再使用するのが便利です。4 つの EGL CRUD コマンドは次のとおりです。

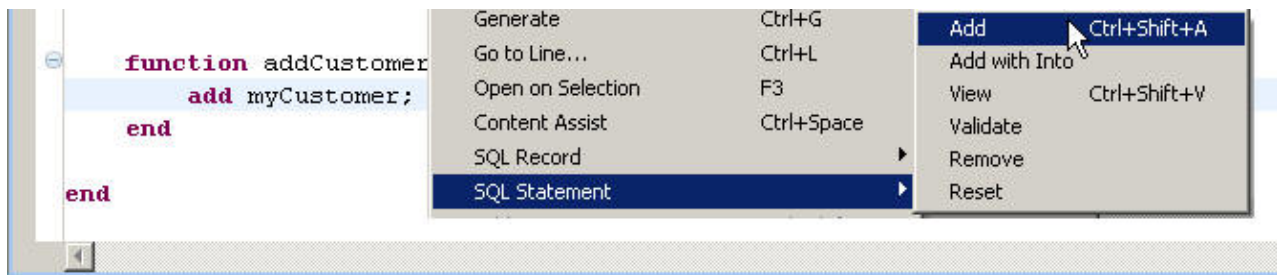
表 1. EGL CRUD コマンドおよびそれと同等の SQL コマンド

EGL コマンド	EGL サンプル	SQL コマンド
add	<code>add myRecord;</code>	INSERT
get	<code>get myRecord;</code>	SELECT
replace	<code>replace myRecord;</code>	UPDATE
delete	<code>delete myRecord;</code>	DELETE

これらのコマンドのいずれかを使用すると、EGL はそのコマンドからデフォルトの SQL ステートメントを生成します。次の **add** ステートメント例にして説明します。

```
add myCustomer;
```

myCustomer レコードにカーソルを置き、右クリックしてから「SQL ステートメント」→「追加」の順にクリックすると、EGL エディターで **add** ステートメントが展開され、SQL コードが明示された状態で表示されます。



```
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)
};
```

実質的な内容は全く変更されません。EGL は、**add** ステートメントから生成する SQL コードを開示しただけです。コードが明示的になったので、EGL コマンドから生成されたデフォルトの SQL を編集できます。

SQL INSERT ステートメントの VALUES 節に、新規データベース行（この場合は myCustomer レコード内のフィールド）に挿入する値がリストされます。**#sql** ブロック内の各値の前にはコロン **:** があり、値が SQL の値ではなく EGL の値であることを示しています。明示的な SQL コードで 사용되는これらの EGL 値は、「ホスト変数」と呼ばれます。

次に、ホスト変数のもう 1 つの例として、**get** ステートメントを使用して、データベースから一連のレコードを削除する関数を挙げます。

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

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

end
```

この関数は、2 バイトの CHAR 型パラメーターを受け取り、そのパラメーターをホスト変数として使用して、これらの 2 つの文字に一致する STATE フィールドを持つレコードを取得します。EGL を明示的な SQL コードと一緒に機能させるためには、ホスト変数を慎重に使用する必要があります。

CRUD 操作の復習

このセクションでは、**get**、**add**、**replace**、および **delete** を使用して、EGL を使用した基本データベース操作を実行します。このセクションはオプションです。4 つの基本 EGL データ・アクセス・ステートメントおよびそれと同等の SQL ステートメントを問題なく使用できる場合は、スキップしてかまいません。

1. CRUDeTest.egl という名前の新規 EGL プログラムを作成します。
 - a. 「プロジェクト・エクスプローラー」ビューで、EGLSQL プロジェクトを右クリックして、「新規」→「プログラム」とクリックします。「新規 EGL プログラム・パーツ」ウィンドウが開きます。
 - b. 「パッケージ」フィールドに、名前 programs を入力します。
 - c. 「EGL ソース・ファイル名」フィールドに、名前 CRUDeTest を入力します。
 - d. 「終了」をクリックします。

新規 EGL プログラムが作成されてエディターで開きます。

2. デフォルトの EGL コードを次のコードで置換します。

```
package programs;

import eglDerby7.data.*;

program CRUDeTest 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
```

この時点で、このプログラムが出力するのは、データベース内の顧客の完全なリストのみです。また、プログラムには、データベースの処理中に EGL に生じたすべて問題を処理する `handleDBException` という名前の関数も含まれています。(エラー処理については、この後のチュートリアルで学習します。) このプログラムを今すぐ実行して、変更前のデータベースの外観を表示できます。

3. プログラムを保存して、生成します。
4. 次のようにして、プログラムを実行します。
 - a. 「プロジェクト・エクスプローラー」ビューで、`EGLSQL/JavaSource/programs` を展開します。
 - b. `CRUDTest.java` ファイルを右クリックし、「実行」→「**Java アプリケーション**」の順にクリックします。

プログラムが実行され、エラーが出なかった場合は、「コンソール」ビューにデータベースの内容が表示されます。「コンソール」ビュー内の結果は次のようになります。

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
```

コンソール内の結果にエラーがある場合は、プログラムを正確にコピーしたことを確認してから、52 ページの『トラブルシューティング』を参照してください。

5. 新規行をデータベースに挿入する関数を、最後の **end** ステートメントの前に追加します。

```
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
```

この関数は、**add** を使用してレコードをデータベースに挿入します。**add** ステートメントの背後にある暗黙 SQL コードを表示するには、行 `add customerToAdd;` のキーワード `customerToAdd` に直接カーソルを置いて右クリックし、「**SQL ステートメント**」→「**追加**」の順にクリックします。これで、ステートメントは次のようになります。

```
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)
};

```

また、レコード変数に直接カーソルを置き、右クリックして「SQL ステートメント」→「表示」とクリックすると、SQL コードを明示せずにプレビューできます。このように EGL コード内で SQL コードを処理するには、EGL エディターのカーソルをステートメント内のどこかに置く必要があります。

- 次に示すように、新規関数をプログラムの `main()` 関数から呼び出します。

```

function main()

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

    // Add a record.
    addCustomer();

end

```

- プログラムを保存し、生成して実行します。出力には、レコードをプログラムに追加する前と後のテーブルの内容が示されます。

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

```

- データベースからレコードを削除する関数を、最後の `end` ステートメントの前に追加します。

```

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. main() 関数で、レコードを追加する関数への呼び出しを、レコードを削除する関数への呼び出しで置換します。 CUSTOMER_ID 列はテーブルの主キーであり、また、テーブルの CUSTOMER_ID の行が既に 50 に設定されているため、レコードを再度追加しようとするエラーが発生します。

```
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. プログラムを保存し、生成して実行します。 出力には、レコードを削除する前と後のテーブルの内容が示されます。

```
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. データベース内のレコードを取得して更新する関数を、最後の **end** ステートメントの前に追加します。

```
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. プログラムの main() 関数で、add、replace、および delete 関数をその順序で呼び出します。

```

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. プログラムを保存し、生成して実行します。出力に、プログラムの開始時、レコードを追加した後、レコードを更新した後、およびレコードを削除した後のテーブルの内容が示されます。

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

これで、CRUDTest.egl ファイルのコードが完成しました。このファイル内にエラーがある場合 (赤の X 記号でマークされている場合) は、56 ページの『演習 2 で完成した CRUDTest.egl ファイル』に記載されているファイルのコードと、作成したコードが一致していることを確認してください。

演習のチェックポイント

この演習では、EGL および SQL を使用したリレーショナル・データベースへのアクセスの基礎を復習しました。

次の概念を学習しました。

- SQLRecord パーツ、およびそれらのパーツとデータベース・テーブルとの関係
- データベース操作を実行する 4 つの基本 EGL ステートメント
- 明示的な SQL
- ホスト変数

4 つの基本 EGL データ・アクセス・ステートメントの詳細については、次のヘルプ・トピックを参照してください。

- add
- delete
- get
- replace

演習 3: open および forEach を使用したカーソルの管理

カーソルはブックマークのように動作する SQL 構成体で、行の位置を指定し、一連の検索結果 (つまり、テーブル行) を一度に 1 つずつステップスルーできるようにします。EGL はカーソルを管理できますが、カーソルを利用するための **open** および **forEach** ステートメントも提供します。

前回の演習で、**delete** ステートメントにキーワード **noCursor** が含まれていたことに気がついたかもしれません。**get** ステートメントがテーブル全体から 1 行以上を選択するのにに対し、通常、**delete** ステートメントは前のステートメントで作成されたカーソルで示される 1 つの行に作用します。**noCursor** キーワードは、そのようなカーソルが存在しないことを示し、**noCursor** ステートメントがデフォルトの **get** ステートメントと同じ方法で行を指定するために **Where** 文節を使用することを指定します。

EGL **open** ステートメントはカーソルを作成し、結果セット、つまり、カーソルが移動可能な 1 行以上のセットを指定します。お客様は、カーソルに直接アクセスする代わりに、他の EGL ステートメントを使用して、カーソルで示されている結果セットから次の行を取得したり、カーソルを使用して一度に 1 行ずつステップスルーしながら、結果セット全体に対する操作を実行します。

カーソルおよび結果セットを EGL で作成するには、**open** ステートメントを使用して 1 行以上を指定し、新規結果セットに名前を付け、カーソルで示される行を入れる一時変数を提供します。

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

この例における tempItem は、データベースの ITEMS テーブルを示す、SQLRecord パーツからの単一レコード変数です。コード for tempItem は、EGL がカーソルを使用して一度に 1 行ずつこのレコードに移動することを指定します。コード open allExpensiveItems は、結果セットに名前を付けます。allExpensiveItems は、厳密に言えば、変数ではなく識別子です。

get ステートメントのように、EGL は **open** ステートメントから SQL SELECT ステートメントを作成するので、お客様はそれを明示して、編集できます。上記の **open** ステートメントの例で、デフォルトの SELECT ステートメントは次のようになります。

```
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
};
```

このデフォルトの Where 文節で指定しているのは、各項目がその前の項目よりも大きい ITEM_ID 値を持つことのみです。(EGL は SQL DECLARE CURSOR および OPEN CURSOR ステートメントを管理しますが、明示的な SQL では表示しません。)

特定の行を選択して結果セットに追加するには、**get** ステートメントの場合のように Where 文節を編集します。例えば、次のようにして、所与の変数よりコストが高い項目のみを選択できます。

```
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
};
```

これで、expensiveItems 結果セットに、500 に設定されている maxCost 変数より大きいか等しい PRICE 列を持つデータベースの行がすべて含まれます。

これらの行にアクセスするには、**forEach** ステートメントを使用します。これは、ループ内のコード・ブロックを実行して、結果セットの各項目にそのコードを適用します。ループを反復するごとに、EGL はカーソルで示された行を一時変数に移動し、カーソルを結果セット内の次の行に進めます。例えば、次のコードは価格が 500 より大きい各項目から値を出力します。

```
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

```

get next を使用すると、結果を手動でステップスルーすることもできます。この場合は、**sysVar.sqlData.sqlcode** の値をテストするなどして、結果セットにまだ結果があるかどうか調べる必要があります (結果セットにさらに結果がある場合、**sysVar.sqlData.sqlcode** の値はゼロになります)。

```

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

```

get ステートメントには、オープン・カーソルおよび結果セットで使用可能な様々な操作が他にもあります。

- **get next** を使用して、前の例のように一時変数を指定するのではなく、結果セットを指定できます。例えば、**get next from expensiveItems** は、結果セットの次の行を一時変数に移動します。このコードは **get next tempItem** と同等です。同様に、**forEach** ステートメントで結果セット名を使用できます。**forEach (from expensiveItems)** は **forEach (tempItem)** と同等です。
- **get current** は、カーソルが現在示している行を、カーソルを次の行に移動することなく取得します。
- **get first** および **get last** は、結果の先頭と末尾の行をそれぞれ取得します。
- **get previous** は **get next** の逆で、カーソルの前の行を取得します。
- **get absolute(position)** は、結果セット内の特定の位置にある行を取得します。例えば、**get absolute(5) tempItem** は結果セット内の 5 行目を取得し、**get absolute(-2) tempItem** は最後から 2 行目を取得します。

- **get relative(position)** は、カーソルの現在位置に対して特定の相対位置にある行を取得します。例えば、**get relative(3) tempItem** は、カーソルの現在位置から 3 行後の行を取得し、**get relative(-2) tempItem** はカーソルの現在位置の 2 行前の行を取得します。**get relative(0) tempItem** は **get current tempItem** と同等です。

get next 以外のこれらの定位置 **get** ステートメントを使用するには、結果セットが **scrollable** であることを指定するか、先頭から末尾まで一度に 1 つの行をステップスルーするのではなく、カーソルの位置を手動で変更する意図があることを示す必要があります。次の例で示すように、**scroll** キーワードを **open** ステートメントに追加します。

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

EGL は、いくつかある条件のいずれかに合致すると、カーソルを自動的に閉じます。例えば、カーソルが結果セットの末尾にある、同じ結果セット上の別の **open** ステートメントにある、あるいは、プログラムの終わりにあるという理由で、**get next** ステートメントで結果が戻らない場合などです。カーソルおよび結果セットの操作が終了した場合、次の例のように、EGL **close** ステートメントを使用して閉じることができます。

```
close expensiveItems;
```

カーソルと結果セットが閉じられると、カーソルに依存していた **get next** やその他のステートメントは機能しなくなります。

結果セットのステップスルー

この演習では、**forEach** ステートメントを使用して結果セットをステップスルーし、データベースの項目のうち、ある価格を上回るものを取得します。

1. **programs** パッケージ内に、**selectItems** という名前の新規プログラムを作成します。
2. プログラムからデフォルトのコードを除去して、プログラムを次のようにします。

```
package programs;

program selectItems type BasicProgram {}

    function main()
    end

end
```

3. データベースへのアクセス中に EGL で発生する可能性のあるすべてのエラーを処理するために、次の関数を追加します。

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

データベース・アクセスで発生したエラーの処理については、この後のチュートリアルで学習します。

4. 次の関数を追加して、最大コストの **Price** パラメーターに基づいてデータベースから行を除去します。

```
function printExpensiveItems(maxPrice Price in)

end
```

Price dataItem パーツは、DECIMAL プリミティブ型のものです。このパーツは、ファイル **eglderbyr7/primitivetypes/data** に定義されています。

5. 関数宣言で Price 変数を追加する際にコンテンツ・アシストを使用しなかった場合は、次の **import** ステートメントをファイル上部の **package** ステートメントの下に追加します。

```
import egllderbyr7.primitivetypes.data.*;
```

コンテンツ・アシストを使用すると、キーワードおよびタイプの名前を、最初の数文字を入力し、Ctrl + スペースを押すことで入力できることを覚えておいてください。コンテンツ・アシストを使用して、Price dataItem パーツなどのタイプを挿入すると、コンテンツ・アシストにより、適切な **import** ステートメントがファイルに追加されます。

6. 新規関数内に、カーソルおよび結果セットを作成するための **open** ステートメントを追加します。カーソルで示される行を保持する一時 Item 変数も必要になります。

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

7. この場合も、Item レコード・タイプを挿入する際にコンテンツ・アシストを使用しなかった場合は、次の **import** ステートメントを追加して Record パーツをスコープに入れます。

```
import egllderbyr7.data.Item;
```

8. **open** ステートメント上にカーソルを置いて右クリックし、「SQL ステートメント」 → 「追加」の順にクリックして、SQL コードを明示します。 **open** ステートメントは次のようになります。

```
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. 明示的な SQL コードの Where 文節を変更して、関数に渡される maxPrice 変数より大きいか等しい EGL.ITEM.PRICE 列の値を持つ行のみを選択するようにします。

```
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  
};
```

maxPrice 変数 が SQL のキーワードや値ではなく、EGL 変数であることを示すために、maxPrice 変数の前にコロンを入れることを忘れないでください。

10. **open** ステートメントの後に、項目についての情報をコンソールに出力する **forEach** ループを追加します。

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

完成した関数は次のようになります。

```

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

```

11. main 関数から関数を呼び出し、maxPrice の値を渡します。

```

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

```

12. プログラムを生成し、実行します。

プログラムにより、データベース内の売上項目のうち、関数に渡された値と等しいかそれより大きい価格の項目名が出力されます。例えば、200 を関数に渡すと、次のような結果になります。

```

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

```

これで、selectItems.egl ファイルのコードが完成しました。このファイル内にエラーがある場合 (赤の X 記号でマークされている場合) は、57 ページの『演習 3 で完成した selectItems.egl ファイル』のファイルに記載されているコードと作成したコードが一致していることを確認してください。

演習のチェックポイント

この演習では、結果セットの 1 行を **open** および **foreach** を使用して一度に処理する方法を学習しました。

結果のリスト全体をレコードの配列に取り出す際に、カーソルを使用して結果セットの 1 行を一度に処理すると、**get** を使用するよりも便利で効率的です。

詳しくは、open および foreach についてのヘルプ・トピックを参照してください。

演習 4: prepare を使用したクエリーの動的構成

EGL **prepare** ステートメントを使用すると、SQL ステートメントが動的に構成されます。

前回の演習で、パラメーターより大きいか等しい価格を持つ品目をデータベースから選択する関数を作成しました。

```

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

```

パラメーターより小さいか等しい価格の品目を返す場合は、どのような関数を使用するのでしょうか。あるいは、最大価格と最小価格との間の品目を返す場合は、どのような関数を使用するのでしょうか。**open** ステートメントで明示的な SQL を使用する場合は、SQL ステートメントの **Where** 文節にある比較がハードコーディングされるため、それぞれの可能性ごとに 1 つ、計 3 つの関数をコーディングしなければなりません。

SELECT ステートメントをより柔軟に作成する方法は、EGL **prepare** ステートメントを使用して、実行時にクエリーをアSEMBルすることです。**prepare** を使用するには、まず SQL コードのテキストを含む **STRING** 変数を作成します。

```

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

```

次に、**prepare** ステートメントを使用して、このストリングを SQL ステートメントに変換します。ステートメントの名前を指定し、オプションで、ステートメントが影響するのがどの **SQLRecord** パーツであるかを示すレコード変数も指定します。

```

tempItem Item;
prepare preparedStatement from selectStatement for tempItem;

```

これにより、**get**、**open**、または **execute** ステートメント内の明示的な SQL の代わりに、準備済みステートメントを使用できるようになります (**execute** については、後の演習でより詳細に学習します)。

```

open expensiveItems for tempItem with preparedStatement;

```

準備済みステートメントでよく発生するエラー

この方法でストリングから SQL ステートメントを作成するのは、SQL に精通したプログラマーの上級テクニックです。EGL は設計時に SQL ステートメントが正確であるかどうか検査しないため、実行時にストリングが正しい SQL ステートメントに解決することを確認する必要があります。以下は、準備済みステートメントの処理でよく生じるエラーのリストです。

ホスト変数

準備済みステートメントでは、明示的な SQL で使用するのと同じ方法でホスト変数を使用できません。次の例を見てください。

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

この場合、実際の SQL ステートメントには、全くそのままのストリング `:maxPrice` が含まれます。そのストリングがホスト変数として使用されて `maxPrice` 変数の値で置換されることはありません。結果は、誤った SQL ステートメントとなります。

代わりに、変数の値をストリングに挿入する必要があります。

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

この場合、`maxPrice` 変数の値は EGL がストリングを作成するときに解決され、次の SQL ステートメントを作成します。

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

後でこの演習で、準備済みステートメントで変数を使用する別の方法を学習します。

スペーシング

SQL ステートメント内のキーワードとキーワードは、スペースで分離してください。例として、複数のストリング値を連結記号 (`::=`) で連結することにより準備された SQL ステートメントを示します。

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

この例からは、次の誤った SQL ステートメントが作成されます。

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

このステートメントには、WHERE の後と ORDER BY の前にそれぞれスペースが必要です。

適切なステートメント

EGL は準備済みステートメントに対して妥当性検査を行いません。ステートメントが正しい SQL であることを確認することは、お客様に任されています。

また、準備済み SQL ステートメントは、明示的な SQL と同様に、一緒に使用する EGL ステートメントに適合している必要があります。例えば、EGL **get** および **open** ステートメントには結果セットが必要です。これらのステートメントのいずれかと一緒に準備済みステートメントを使用するには、SQL コードが結果セットを返す必要があります。このため、SQL INSERT または DELETE ステートメントを準備し、そのステートメントを **get** または **open** と一緒に使用することはできません。

演習 4A: 準備済みステートメントの使用

この演習では、前回の演習で作成したプログラムを、2 つのパラメーター (そのうちの 1 つまたは両方を NULL にすることが可能) を受け入れるように変更します。この関数により、非 NULL のパラメーターに基づいてクエリーが作成され、ある一定の範囲の価格の品目が戻ります。

1. `selectItems.egl` ファイルを開きます。
2. NULL 可能な 2 つの Price パラメーターを受け取る `printItemRange` という名前の関数を追加します。

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

end
```

パラメーター・タイプの後の疑問符 (?) は、変数が NULL 値を受け取ることができることを示しています。このようにして、関数は結果の範囲について最小と最大のいずれかまたは両方を受け取ることができます。

3. SQL コードを保持して SELECT ステートメントの開始を SQL コードに挿入する、STRING 変数を作成します。

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

次のステップは、適切な Where 文節がどのような分節であるかを判別することです。次の 4 つの状況が考えられます。

- 両方のパラメーターが NULL である。この場合、関数はゼロより大きい等しい価格の品目をすべて出力するため、Where 文節は where EGL.ITEM.PRICE > 0 になります。
- 両方のパラメーターが指定されている。この場合、関数は SQL BETWEEN キーワードを使用して、パラメーターとパラメーターの間、またはパラメーターと等しい値を出力します。Where 文節は、where EGL.ITEM.PRICE between :minPrice and :maxPrice になります。
- 最大パラメーターのみが指定されている。この場合、関数は最大より小さい等しい価格の品目をすべて出力し、where EGL.ITEM.PRICE <= :maxPrice になります。
- 最小パラメーターのみが指定されている。この場合、関数は最小より大きい等しい価格の品目をすべて出力し、where EGL.ITEM.PRICE >= :minPrice になります。

これを作成する方法はいくつかありますが、単純な方法として、3 つのネストされた if ステートメントを使用する方法を示します。

4. 次のコードを関数に追加して、Where 文節を完成させます。

```
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
```

5. ORDER BY 節を使用してステートメントを完成させます。

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

6. SQL ステートメントが正しいことを確認するために、コンソールに出力します。

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

7. 使用するステートメントを準備します。これは、準備済みステートメントがアクセスするテーブルから作成されるレコードを参照して行います。

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

8. **open** ステートメントで、明示的な SQL の代わりに準備済みステートメントを使用します。

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

9. クエリーの結果を出力します。

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

完成した関数は次のようになります。

```
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
```

10. プログラムの main 関数から関数を呼び出します。

```

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

```

11. プログラムを生成し、実行します。

上記の例のパラメーター値を使用して、関数は価格が 50 から 500 の間の品目を出力します。

```

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

```

次の例に示すように、パラメーター値を編集して、NULL 値を含むさまざまな範囲を試行できます。

```

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

```

値として 500 と NULL を渡すと、次のような結果が生じます。

```

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

```

これで、selectItems.egl ファイルのコードが完成しました。このファイル内にエラーがある場合 (赤の X 記号でマークされている場合) は、58 ページの『演習 4A で完成した selectItems.egl ファイル』のファイルに記載されているコードと、作成したコードが一致していることを確認してください。

演習 4B: prepare ステートメント内のホスト変数

前の例では SQL ステートメントは、Where 文節内の変数が固定されていたという点で制限がありました。どの変数が使用されるかわかる前であっても、準備済みステートメントを作成することができます。このタイプのステートメントはより複雑ですが、ステートメントを準備しておいて後で詳細を決めたい場合には柔軟性があります。

前の例では、変数をクエリー・ストリングの残りの部分と連結するために、変数の値を解決しなければなりませんでした。

```

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

```

準備済みステートメントを作成する際に、疑問符 (?) をステートメントに置いて、その値が後で埋められることを示すこともできます。

```
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;
```

これで、準備済みステートメントに 2 つのホスト変数を置く場所ができました。これらの場所は、**BETWEEN** 分節内の疑問符 (?) によって示されています。準備済みステートメントを使用する準備ができたら、**using** ステートメントを使用して値を挿入します。例えば、価格が 5 から 500 の間の行を検索するには、リテラルの 5 と 500 をステートメントに渡します。

```
open expensiveItems2 for tempItem with preparedStatement2
    using 5, 500;
```

もちろん、**using** キーワードと一緒に変数を使用することもできます。

```
open expensiveItems2 for tempItem with preparedStatement2
    using minPrice, maxPrice;
```

このような方法で準備済みステートメントでホスト変数を使用すると、ステートメントを 1 度アセンブルしておいて、異なる値で何度も呼び出す際に便利です。例えば、次の関数は 3 つの異なる範囲内にある値を出力するために、3 回同じ準備済みステートメントを使用します。

```
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

end
```

ホスト変数をこのように使用すると、前のセクションのプログラムを再作成でき、起こり得るケースごとにステートメントを変更することなく 1 つの準備済みステートメントを使用できます。

1. 前のセクションと同様に、selectItems.egl ファイルで、最大および最小パラメーターを受け入れる新規関数を追加します。

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

end
```

この関数でステートメントを準備してから、後で変数を挿入します。

2. 準備済みステートメントを保持するストリング変数を作成し、**prepare** ステートメントを使用して準備済みステートメントを準備します。

```
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;
```

この準備済みステートメントに 2 つのパラメーターを渡して、検索の境界を示します。最大価格が指定されていない場合は、検索の上限を知っておく必要があります。なぜなら、ステートメントを準備した後で `EGL.ITEM.PRICE between ? and ?` 分節を `EGL.ITEM.PRICE >= ?` に変換することはできないからです。したがって、上限として使用するテーブル内の最高価格を調べておく必要があります。この上限は、`SQL MAX()` 関数で簡単に調べることができます。

3. `MAX()` 関数を使用してそれをレコード変数の `Price` フィールドに入れて `Item` 変数を作成し、データベース内の最も高額の商品の価格を取得します。

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

これで、`mostExpensiveItem` レコードに、`Items` テーブル内の最高価格が含まれます。

4. どのパラメーターが指定されたかを判別し、それに従って準備済みステートメントを使用します。

```
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

end
```

5. **forEach** を使用して、結果を出力します。

```

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

```

完成した関数は次のようになります。

```

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. main 関数を変更して、古い関数の代わりに新しい関数を呼び出します。

```

function main()
  minPrice, maxPrice Price?;
  minPrice = 50;
  maxPrice = 500;

```

```

try
    printItemRange2(minPrice, maxPrice);
onException(exception SQLException)
    handleDBException(exception);
end
end

```

7. NULL 値を含むさまざまな値の minPrice および maxPrice を使用する新規プログラムを保存し、生成して実行します。結果は前の関数と同じですが、関数は異なる方法で結果に達しました。

これで、selectItems.egl ファイルのコードが完成しました。このファイル内にエラーがある場合 (赤の X 記号でマークされている場合) は、60 ページの『演習 4B で完成した selectItems.egl ファイル』に記載されているファイルのコードと、作成したコードが一致していることを確認してください。

演習のチェックポイント

有効な準備済みステートメントの作成は複雑ではありますが、作成されたステートメントは明示的な SQL でハードコーディングされたステートメントよりも柔軟性があります。後の演習で、SQL ステートメントをカスタマイズするもう 1 つの方法である、SQLRecord パーツにデフォルトの SQL コードを設定する方法を学習します。

準備済みステートメントについて詳しくは、「prepare」を参照してください。

演習 5: テーブル結合を使用した、より複雑なデータの取得

これまでは、それぞれのデータ・アクセス関数が 1 つのテーブルからのみ行を取得していました。この演習では、EGL で SQL テーブル結合を使用して、一度に複数のテーブルを処理する方法を学習します。

SQL テーブル結合の詳細な説明は、このチュートリアルでは割愛します。簡単に述べると、テーブル結合は、2 つ以上の関連テーブルから結果セットを組み立てます。

例えば、データベース内の CUSTOMERS テーブルには、主キーとしてカスタマー ID 番号が含まれ、それと一緒に顧客の名と姓 (および他のいくつかの列) が含まれています。

表 2. CUSTOMER テーブルのサンプル・データ

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

ORDERS テーブルには、主キーのオーダー ID 番号に加え、オーダー数が含まれています。このテーブルには、発注した顧客の ID 番号も含まれており、その列は「外部キー」になっています。

表 3. ORDERS テーブルのサンプル・データ

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

この場合、CUSTOMER_ID 列が 2 つのテーブル間の関係を作成するため、テーブル結合の適切な候補となります。この演習では、これら 2 つのテーブルをカスタマイズされた SQLRecord パーツで結合し、顧客と顧客が行ったオーダーとを突き合わせる結果セットを生成します。

テーブル結合は複雑なデータベース操作であるため、データベースの動作に精通している必要があります、また出力を慎重にテストする必要があります。この演習でわかるように、必ずしも期待どおりのデータを得られるわけではありません。

1. 新規 SQLRecord パーツを保持する新規 EGL ソース・ファイルを作成します。
 - a. 「プロジェクト・エクスプローラー」ビューで、EGLSQL プロジェクトを右クリックし、「新規」→「EGL ソース・ファイル」とクリックします。
 - b. 「新規 EGL ソース・ファイル」ウィンドウで、「パッケージ」フィールドに名前として data を入力します。
 - c. 「EGL ソース・ファイル名」フィールドに、records と入力します。
 - d. 「終了」をクリックします。

新規 EGL ソース・ファイルが作成されてエディターで開きます。

2. 新規ファイルの **package** ステートメントの下に、CUSTOMER と ORDERS の両方のテーブルを結合する新規 SQLRecord の名前を入力します。

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

このレコードの 1 行目はもう一方の SQLRecord パーツと同じで、レコードの名前と SQLRecord ステレオタイプを指定しています。2 行目は、このレコードが関連するテーブルを、**tableNames** プロパティの値として指定しています。前の演習で使用した単一テーブル・レコードでは、このプロパティに、Orders レコードの ORDERS テーブルなど 1 つのテーブルのみがリストされました。

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

作成している OrderCustomerJoin レコードでは、2 つのテーブルからデータを取得するために、**tableNames** プロパティに 2 つのテーブルがリストされる必要があります。わかりやすくするために、レコードにそれぞれのテーブル名の別名 (ORDERS を表す「O」と CUSTOMERS を表す「C」) を含めます。後で説明しますが、これらの別名により、レコード内のテーブル名を簡単に参照できるようになります。

これでレコード・パーツの基本情報が指定されました。残りの情報は EGL によって埋められます。ただし、EGL がどのデータベースから接続情報を取得するかを最初に指定する必要があります。

3. 次のようにして、EGL のデフォルトの SQL データベースとして Derby データベースを設定します。
 - a. 「ウィンドウ」→「設定」とクリックします。
 - b. 「設定」ウィンドウで、「EGL」を展開して「SQL データベース接続」をクリックします。
 - c. 「接続」リストで、使用するデータベース接続 (EGLDerbyR7 または EGLDerbyR71) を選択します。
 - d. 「OK」をクリックします。
4. records.egl ファイルに戻り、レコード名の上にカーソルを置きます。
5. OrderCustomerJoin レコードの名前の上にカーソルを置いて右クリックし、「SQL レコード」→「SQL を検索」とクリックします。EGL SQL 取得機能
6. パスワード・プロンプトで、ユーザー名およびパスワードのデフォルト値をそのまま使用します。サンプル・データベースでは、特定のユーザー名またはパスワードは必要ありません。EGL SQL 取得機能により、SQLRecord パーツの残りの部分、例えば、テーブル内の行や **keyItems** プロパティに対するフィールドなどが作成されます。

```

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

```

各フィールドの **column** プロパティの値が、**tableNames** プロパティで指定したテーブルの別名で始まることに注意してください。この別名は、特定のフィールドが関連するテーブルを追跡するのに便利です。

7. records.egl ファイルを保存しますが、開いたままにします。
8. printCustomerOrders という名前のプログラム・パッケージ内に、新規プログラム・パーツを作成します。
9. 新規プログラムからデフォルト・コードを除去します。
10. プログラムの main 関数で、上で作成した OrderCustomerJoin レコードに基づいて配列変数を作成します。

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

Ctrl + スペースを押すことにより、コンテンツ・アシストを使用してレコード・パーツ・タイプを挿入できることを覚えておいてください。コンテンツ・アシストを使用しない場合は、手動で **import** ステートメントをファイル上部の **package** ステートメントのすぐ下に追加します。

```
import data.OrderCustomerJoin;
```

11. **open** および **forEach** ステートメントを使用して、データベースから行を取得し、値をコンソールに出力します。

```

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

```

12. プログラムを保存し、生成して実行します。 テーブル結合で陥りやすいエラーについての知識がある場合は、このテーブル結合のエラーに気づき、なぜ出力があまり有用でないかにお気づきになると思われます。

```

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
...

```

データベースは、テーブル間でロジックの相互参照を行いませんでした。代わりに、CUSTOMERS テーブル内の各行と ORDERS テーブル内の各行とを可能な限りの組み合わせでペアにしました。この結果、それぞれの顧客があらゆるオーダーを行ったものとして記録されましたが、もちろんこれは誤っています。 **open** ステートメントの背後にある SQL コードを明示すると、SQL が Where 文節を使用することなく、両方のテーブルからすべての行を選択していることがわかります。

```

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
  };

```

意味を成すようにテーブルを結合するには、単にテーブルの結合を返すのではなく、論理的にテーブルをマージする方法をデータベースに指示する必要があります。明示的な SQL コードを編集してこれを行うことができますが、この場合、レコードを使用するたびに編集を行わなければなりません。そうしないと、不正確なデータを取得する危険が生じます。代わりに、デフォルトの選択条件をこのレコードに設定し、暗黙的な SQL コードが常に意味のあるテーブル結合を実行するようにします。

13. **open** ステートメントの背後にある SQL コードを明示した場合は、カーソルをレコード名の上に置いて右クリックし、「SQL ステートメント」 → 「削除」とクリックして、再度コードを暗黙的にします。
14. records.egl ファイルで、レコード・パーツの定義に戻ります。
15. **defaultSelectCondition** プロパティをレコード・パーツの定義に追加します。

```

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

```

defaultSelectCondition プロパティでは、ユーザーがこのレコードで **get** または **open** を使用するときには用いられる、暗黙的な SQL コードとデフォルトの明示的な SQL コードの Where 文節を指定します。

16. **#sqlCondition{}** ブロック内で、デフォルトの「condition」テキストの代わりに、これら 2 つのテーブル間の SQL テーブル結合で使用する正しい Where 文節を入力します。この際、完全なテーブル名の代わりにテーブル別名を使用するようにします。

```

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

```

キーワード WHERE を追加する必要はありません。この場合、EGL が自動的に SQL コードに挿入します。

17. ファイルを保管します。ここで、**open** ステートメントの背後にある SQL コードを明示すると、次のようになります。

```
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
  };
```

これで、CUSTOMER と ORDERS の両方のテーブルに同じカスタマー番号がある行のみを返すように制限する Where 文節が SQL コードに含まれました。

18. プログラムを保存し、生成して実行します。

有用なテーブル結合により、このプログラムの結果として、それぞれの注文を行った顧客が表示されます。

```
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
```

これで、この演習で使った 2 つのファイルのコードが完成しました。いずれかのファイル内に赤い X 記号がマークされたエラーが表示される場合は、ご使用のコードが次のコード 62 ページの『演習 5 で完成した printCustomerOrders.egl および records.egl ファイル』と一致するか確認してください。

演習 6: 任意の SQL コードの実行

この演習では、EGL でより柔軟に SQL コードを処理する方法を学習します。

ここまでは、EGL ステートメントと一致する SQL ステートメントを使用するように制限されていました。例えば、**get** ステートメントに関連した SQL コードを編集できるにもかかわらず、その SQL コードは SELECT ステートメントに限定されていました。例えば、**get** ステートメントの明示的な SQL コードには、SQL DELETE ステートメントを含めることができませんでした。明示的な SQL は編集可能ですが、その SQL コードは EGL キーワードに適したものでなければなりませんでした。

それでは、直接対応する EGL キーワードがない SQL ステートメントを使用するにはどうするのでしょうか。EGL キーワードに関連付けられていない SQL コードを実行するには、ワークベンチ・ツールを直接使用するか、EGL **execute** ステートメントを使用します。

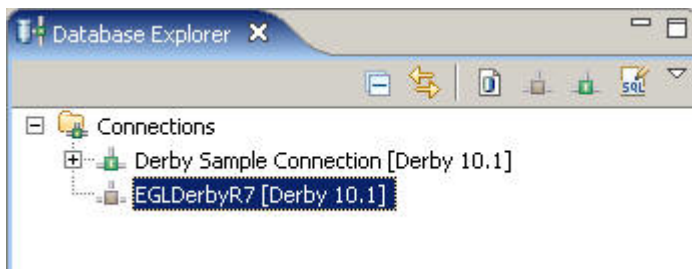
SQL ビルダーを使用した SQL ステートメントの作成

ワークベンチには、EGL アプリケーションの外にあるデータベースの処理に役立ついくつかのツールが用意されています。このセクションでは、SQL ビルダーを使用してサンプル・データベースをより直接的に処理します。

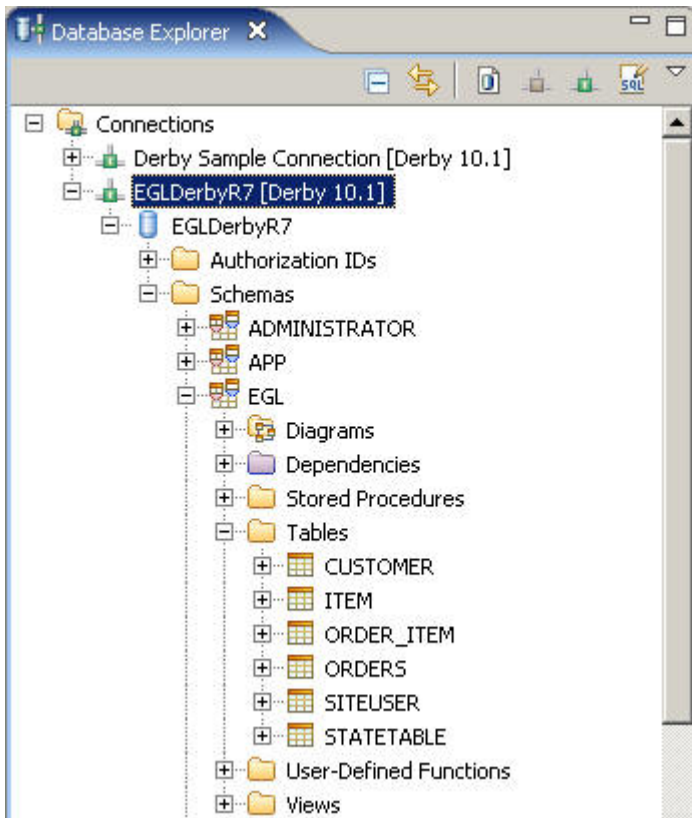
1. 次のようにして、データ・パースペクティブに切り替えます。
 - a. 「ウィンドウ」 → 「パースペクティブを開く」 → 「その他」の順にクリックします。
 - b. 「パースペクティブを開く」ウィンドウで、「データ」をクリックします。
 - c. 「OK」をクリックします。

データ・パースペクティブが開き、データベースの処理に使用するビューが表示されます。大抵の作業は、「データベース・エクスプローラー」ビューで行います。

2. 「データベース・エクスプローラー」ビューで、「接続」を展開します。「データベース・エクスプローラー」ビューに、データベースへのプロジェクトの接続として EGLDerbyR7 が、テストで 사용되는サンプルの Derby 接続と一緒に表示されます。



3. EGLDerbyR7 接続を右クリックし、「再接続」をクリックします。これで、「データベース・エクスプローラー」ビューがデータベースに接続されたので、データ・パースペクティブでツールを使用してデータベースを処理できます。ただし、Derby は 1 回に 1 つの接続しか許可しないため、「データベース・エクスプローラー」ビューが接続されている間は、EGL プログラムはデータベースに接続できません。
4. 「データベース許可」ウィンドウで、「OK」をクリックします。
5. 「EGLDerbyR7」 → 「EGLDerbyR7」 → 「スキーマ」 → 「EGL」 → 「テーブル」の順に展開します。これで、「データベース・エクスプローラー」ビューに、このチュートリアルに関するデータベース内のテーブルがリストされます。他のテーブルはメタデータで、このチュートリアルでは重要ではありません。



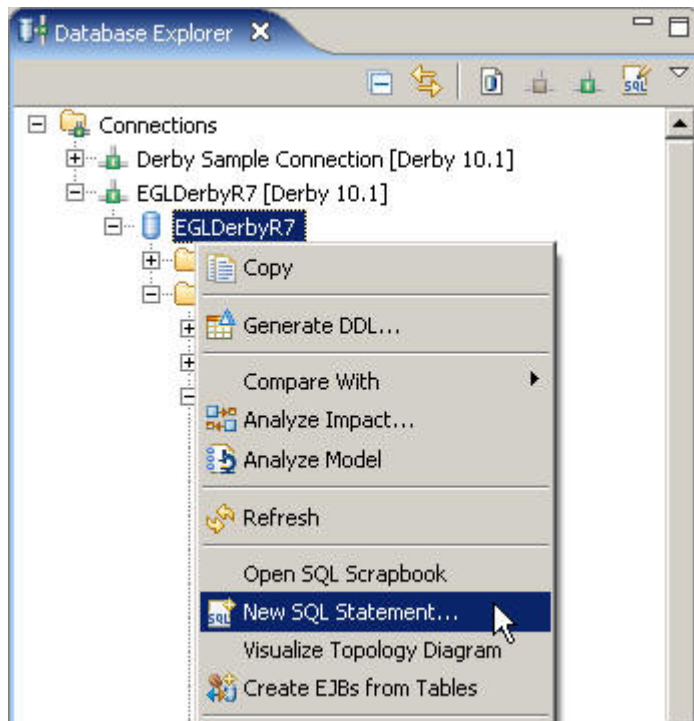
- CUSTOMERS テーブルを右クリックして、「データ」→「サンプル・コンテンツ」とクリックします。「データ出力」ビューに、CUSTOMERS テーブルのデータが表示されます。

Sample Contents

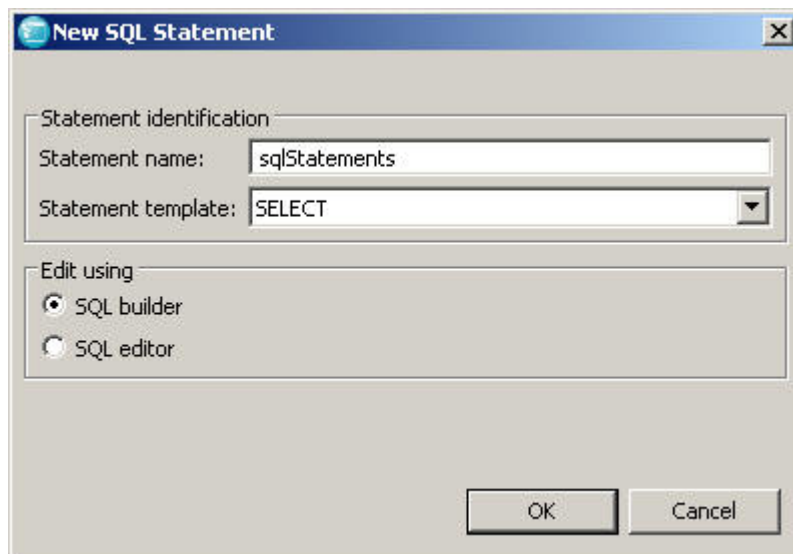
CUSTOMER...	FIRST_NAME	LAST_NAME	PASSWORD	PHONE	EMAIL_AD...	STREET	APARTM
1	Fred	Filibuster	dsds	(201)652-3...	f_filiBust@...	14 Maple A...	1A
2	Andy	Lundquist	bbb	(221)545-5...	alundy@yn...	1338 Dean ...	2B
3	Billie	Kingman	ccc	(261)898-4...	bjPride@ec...	14 Mayberr...	TGIF
4	Francine	Liebowitz	aa	(414)923-2...	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...	

「データ出力」ビューは、EGL **get** ステートメントの背後にある暗黙 **SELECT** ステートメントのように、デフォルトの **SELECT** ステートメントを実行しています。より直接的にデータベースを処理し、EGL プログラムに追加する前に **SQL** コードをテストするために、**SQL** ビルダーを使用して **SQL** コマンドを対話式に作成し、実行できます。

7. 「データベース・エクスプローラー」ビューで、青のデータベース・アイコンで示されている EGLDerbyR7 データベースを右クリックし、「新規 SQL ステートメント」をクリックします。



8. 「新規 SQL ステートメント」ウィンドウで、ステートメントに `sqlStatements` と名前を付けます。
9. 「ステートメント・テンプレート」フィールドを `SELECT` に設定したままにし、「使用中のものを編集」ラジオ・ボタン・グループを「SQL ビルダー」に設定したままにします。



10. 「OK」をクリックします。SQL ビルダーが開きます。このエディターから、SQL ステートメントを作成し、アプリケーションに追加する前にテストできます。
11. 例として、カスタマー・レコードのデフォルトの SQL ステートメントを SQL ビルダーに貼り付けます。EGL エディターで明示的な SQL からこのステートメント（またはその他のなんらかの SQL ステートメント）をコピーすることも、次のコードを使用することもできます。

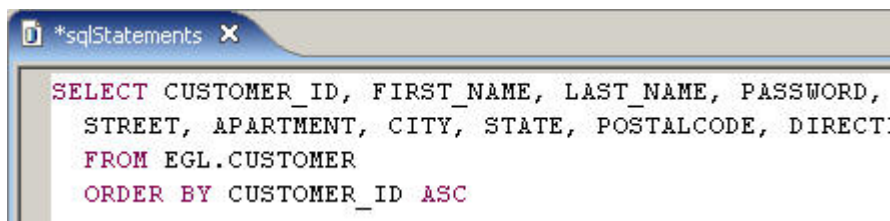
```

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

```

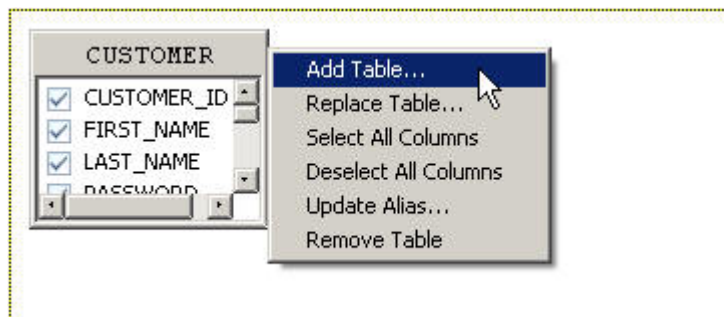
SQL ビルダーに、SQL コードがさまざまな方法で表示されます。

- SQL ビルダーの最上部にあるコード・エディターを使用すると、EGL エディターと同じ方法で SQL ステートメントを編集できます。ただし、このエディターには SQL ステートメント用の特別なオプションがあります。例えば、デフォルトの SQL ステートメントを EGL コードからこのエディターに貼り付ける際に、SQL エディターはステートメントを単純化し、再フォーマットします。



また、この SQL エディターには、SQL ステートメントのためのコンテンツ・アシストが用意されています。EGL コンテンツ・アシストと同様に、Ctrl + スペースを押して、候補リストを表示できます。

- 最上部から 2 つ目のセクションには、ステートメントに関係するテーブルとその列のグラフィック・ビューが表示されます。SELECT ステートメントから列を選択または削除するには、列の横にあるチェック・ボックスをクリアします。また、ステートメントに新規テーブルを追加するには、エディター域を右クリックし、「表の追加」をクリックします。テーブルを右クリックすると、テーブルで使用されるオプションのリストも表示できます。



- 最下部のセクションには、ステートメントがテーブル形式のビューで表示されます。テーブルにあるテンプレートを使用して、ステートメントの編集に役立てることができます。

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

例えば、次のようにして、エディターで SQL ビルダー内のテーブル域を使用して、Where 文節を SELECT ステートメントに追加できます。

12. SQL ビルダーの「条件」タブを開きます。
13. 次のようにして、「条件」タブで新規 Where 文節をステートメントに追加します。
 - a. 「条件」タブで、「列」列の空の行をクリックします。
 - b. 「列」の下にあるブランクのセルで、CUSTOMER.POSTALCODE を選択します。
 - c. 「演算子」で、LIKE を選択します。
 - d. 「値」で、次のコードを入力します。

1%

% 記号 (%) は、SQL のワイルドカードです。この Where 文節により、先頭が 1 の郵便番号を持つ顧客にステートメントが制限されます。

テーブル域では選択条件は次のように示されます。

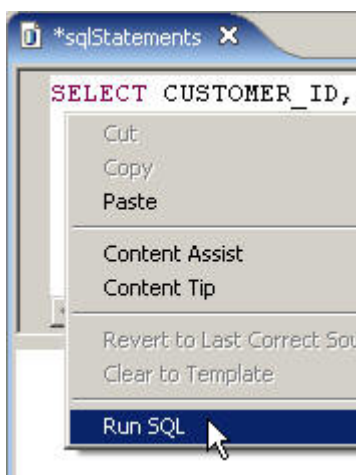
Columns	Conditions	Groups	Group Conditions
Column	Operator	Value	AND/OR
POSTALCODE	LIKE	'1%'	

条件の追加が完了し、フォーカスを他の場所に設定すると、SQL ビルダーによってエディター内の SELECT ステートメントのコードが更新されます。

```

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

14. SQL ビルダーの最上部にあるコード・エディターを右クリックして、「SQL の実行」をクリックします。



「データ出力」ビューに、ステートメントの結果が表示されます。この場合は、先頭が 1 の郵便番号を持つ顧客が表示されます。

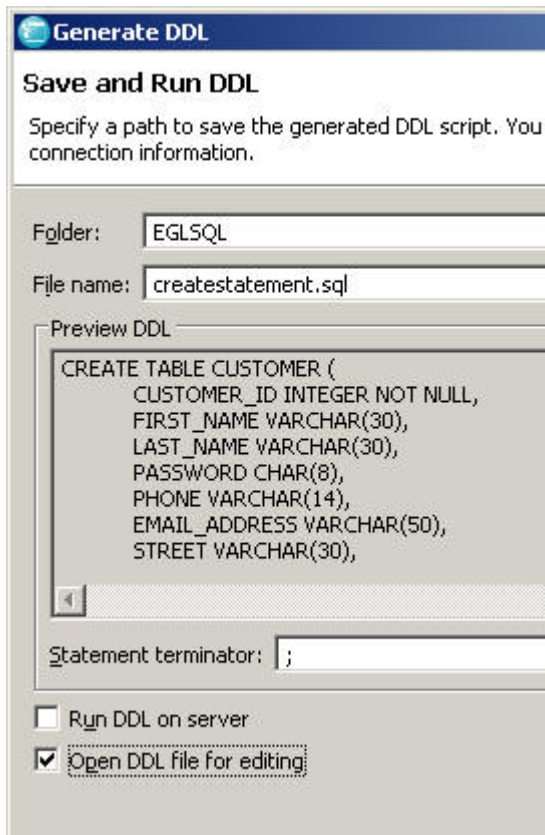
Messages	Parameters	Results	Profiling Data	
CUSTOMER...	FIRST_NAME	LAST_NAME	PASSWORD	PHONE
4	Francine	Liebowitz	aa	(414)923-2...
8	Sergey	Sharov	ds	(413)452-1...

SQL に精通している場合は、独自の SQL ステートメントをここに書き込み、アプリケーションで使用する前にそれをテストできます。

15. SQL ビルダーでの作業が完了したら、コードをファイルに保存するか破棄します。
16. より有用な例として、次のようにして、CUSTOMER テーブルの設計をコピーして新規テーブルを作成するための SQL ステートメントを作成します。
 - a. 「データベース・エクスプローラー」ビューで、CUSTOMER テーブルを右クリックしてから、「DDL の生成」をクリックします。
 - b. 「DDL の生成」ウィンドウで、「CREATE ステートメント」チェック・ボックスを選択し、他のチェック・ボックスをクリアします。



- c. 「次へ」をクリックします。
- d. 「オブジェクト」ページで、「すべて選択」をクリックします。
- e. 「次へ」をクリックします。
- f. 「フォルダー」フィールドで、EGLSQL プロジェクトを選択します。
- g. 「DDL ファイルを保管して実行」ページで、ファイルに createtable.sql と名前を付けます。
- h. 「サーバー上で DDL を実行」チェック・ボックスをクリアします。
- i. 「編集のために DDL ファイルを開く」チェック・ボックスを選択します。「DDL を保管して実行」ページは次のようになります。



j. 「次へ」をクリックします。

k. 「終了」をクリックします。

SQL ステートメント・エディターで、新規 SQL ステートメントが開きます。このエディターは SQL ビルダーと機能が似ていますが、コード編集域しかなく、追加のグラフィック域はありません。

17. ステートメントを編集して、作成されるテーブル名を、CUSTOMER ではなく EGL.CUSTOMERTEMP に変更します。編集済み SQL コードは、次のようになります。

```
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. ステートメントを保存します。

19. 「データベース・エクスプローラー」ビューで EGLDerbyR7 データベースを右クリックし、「切断」をクリックして、データベースを切断します。

これで、データベースに新規テーブルを作成する SQL ステートメントを用意できました。しかし、EGL には SQL CREATE ステートメントと同等のステートメントがなく、また、このステートメントを **get** な

どのステートメントのための明示的な SQL に挿入することもできません。、**get** にはデータベースからの結果セットが必要なためです。次のセクションでは、**execute** を使用して、この SQL ステートメントを EGL で実行します。

execute の使用

新規 CREATE ステートメントの実行は、データ・パースペクティブでツールを用いて行うことができますが、ご使用の EGL アプリケーション内でカスタム SQL コードを実行したい場合があります。言うまでもなく、EGL アプリケーションにカスタマイズされた SQL コードを挿入する場合は必ずそれをテストし、予測どおりに機能するか確認する必要があります。

EGL **execute** ステートメントを使用すると、SQLRecord 変数を扱うことなく、SQL コードを実行できます。**execute** を使用して、CREATE、ALTER、および DROP TABLE、INSERT、DELETE、および UPDATE など、さまざまな SQL ステートメントを実行できます。例えば、次のようにしてテーブルをデータベースから削除できます。

```
execute #sql{
    DROP TABLE OLD_TABLE;
}
```

execute ステートメントは、一部の SQL ステートメントをサポートしません。特に、SELECT と OPEN はサポートしません。**execute** と一緒に使用した場合に機能する SQL ステートメントと機能しない SQL ステートメントのリストは、「SQL での execute についての考慮事項」を参照してください。

execute を使用して、ストアド・プロシージャを呼び出すこともできます。

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

Derby は、大多数のデータベース管理システムと同じ方法ではストアド・プロシージャをサポートしません。したがって、このチュートリアルではストアド・プロシージャを扱っていません。このセクションでは、その代わりに、**execute** を使用して、データベース内にテーブルを作成し、SQLRecord パーツを使用せずにそこにデータを取り込みます。

1. EGL パースペクティブに戻ります。
2. パッケージ名 `libraries` 内の `CustomerTableOperations.egl` という名前のファイルに、新規ライブラリー・パーツを作成します。
3. `execCreateTable`、`execInsertIntoTable`、および `execDropTable` という名前のライブラリーに新規関数を挿入します。各関数はパラメーターとして `StatusRec` レコード変数を受け取ります。

```
function execCreateTable(status StatusRec inOut)
end

function execInsertIntoTable(status StatusRec inOut)
end

function execDropTable(status StatusRec inOut)
end
```

`StatusRec` レコード変数は、データベース操作の成功または失敗についての情報を記録します。

4. パラメーターを追加する際にコンテンツ・アシストを使用しなかった場合は、次の **import** ステートメントをファイル上部の **package** ステートメントのすぐ下に追加します。

```
import egl Derby7.StatusRec;
```

5. `execCreateTable` 関数で **execute** ステートメントを使用して、前のセクションで作成した `CREATE TABLE` ステートメントを実行します。

```
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
```

この関数はテーブルを作成し、次に、エラーがあれば、データベースからデータ・パーツおよびロジック・パーツと一緒に作成される `HandleSuccess` 関数を使用してエラーを処理します。

6. これらのコードを追加する際にコンテンツ・アシストを使用しなかった場合は、次の **import** ステートメントをファイルの先頭に追加します。

```
import egllderbyr7.ConditionHandlingLib;
```

7. `execDropTable` 関数に、データベースからテーブルを削除する **execute** ステートメントを追加します。

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

end
```

8. `execInsertIntoTable` 関数に、`CUSTOMER` テーブルのレコードを `CUSTOMERTEMP` テーブルに移動する **execute** ステートメントを追加します。

```
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
```

このコードは、`CUSTOMER` の各レコードを `CUSTOMERTEMP` テーブルにコピーするのみです。必要があれば、`Where` 文節を追加して、特定のレコードのみを選択してコピーするようにできます。

9. ライブラリーを保存して、生成します。

10. programs パッケージにある duplicateTable.egl という名前のファイルに、新規プログラム・パーツを作成します。
11. 新規プログラムで、新規テーブルを作成する関数を呼び出し、レコードを挿入します。

```
package programs;

import eglDerby7.StatusRec;
import libraries.CustomerTableOperations;

program duplicateTable type BasicProgram {}

    status StatusRec;

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

end
```

12. プログラムを生成し、実行します。
13. データ・パースペクティブに戻り、「データベース・エクスプローラー」ビューでデータベースに再接続します。
14. 前のセクションで行ったように、新規テーブルでデータのプレビューを表示し、新規テーブルとデータを確認します。
 - a. 「EGLDerbyR7」 → 「EGLDerbyR7」 → 「スキーマ」 → 「EGL」 → 「テーブル」の順に展開します。
 - b. CUSTOMERTEMP テーブルを右クリックして、「データ」 → 「サンプル・コンテンツ」とクリックします。

これで、新規テーブルのデータを表示できます。

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...

ここで、他のステートメントを作成して新規テーブルを処理したり、execDropTable 関数を呼び出してそれらのテーブルをデータベースから削除したりできます。例えば、CREATE TABLE ステートメントと一緒に作成された ALTER TABLE および CREATE INDEX ステートメントを実行することができます。しかし、この場合に、Derby では主キー制約のために重複した名前は許可されないため、制約とインデックスの名前を変更する必要があります。「データベース・エクスプローラー」ビューでデータベースを切断してから、データベースにアクセスする EGL コードを実行することを忘れないでください。

これで、この演習で使った 2 つのファイルのコードが完成しました。いずれかのファイル内に赤い X 記号がマークされたエラーが表示される場合は、ご使用のコードが次のコード 64 ページの『演習 6 で完成した CustomerTableOperations.egl および duplicateTable.egl ファイル』と一致するか確認してください。

演習のチェックポイント

この演習では、ワークベンチに用意されているいくつかのデータベース管理ツールについて学習し、また、それらのツールを **execute** ステートメントと組み合わせて、EGL アプリケーション内でさまざまな SQL ステートメントを実行する方法を学習しました。

ワークベンチに用意されているデータ・アクセス・ツールについて詳しくは、「SQL ステートメントの処理」を参照してください。**execute** ステートメントについて詳しくは、「execute」を参照してください。

演習 7: 例外処理およびロールバック

データ・ソースにアクセスするアプリケーションでエラーを適切に処理することは、誤ったデータの取得や挿入を避けるためにも、アプリケーションとデータ・ソース間で不整合が発生するのを避けるためにも、特に重要です。この演習では、リレーショナル・データベースへのアクセス時に発生するエラーに対処するために使用可能な、いくつかのツールについて学習します。

SQL データ・アクセスでの例外処理

このチュートリアル例では、**try** ブロック内にデータ・アクセス・ステートメントを置くケースが多かったことにお気づきだと思います。

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

try ブロックにコードを置くと、EGL はこのコードの実行を通常どおりに試みます。EGL でエラーが発生した場合、EGL は **try** ブロック内でコードの実行を停止し、**try** ブロックに残っているステートメントはすべてスキップして、直接 **onException** ブロックに移動します。**onException** ブロック内に、エラーからのリカバリー、問題の修正、またはログ・ファイルへのエラー情報の書き込みを行うためのコードを含めることができます。

try ブロック内で EGL にエラーが生じると、エラー情報を提供する例外レコードも作成されます。このレコードは **onException** ブロックに渡されます。**onException** ブロック内で、そのレコード内の情報を使用して、エラーの原因を判別し、リカバリーできます。上記の例で、**onException** ブロックは、例外レコードを、エラーを処理する関数に渡します。

例外レコードは、エラー・タイプに応じて、いくつかのステレオタイプの 1 つを持っている可能性があります。上記の例で、**onException** ブロックは、SQLException ステレオタイプを持つ例外レコードを必要とします。このステレオタイプは、SQL エラーおよびその他のリレーショナル・データベース用です。NULL 値の参照などの、異なるタイプのエラーが EGL で生じた場合は、異なるタイプの例外レコード (NULL 値の参照の場合は、NullPointerException ステレオタイプを持つ例外レコード) が作成されます。

次の例のように、異なるタイプのエラーを異なる方法で処理するために、単一の **try** ブロックの後に複数の **onException** ブロックを置くことができます。

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

```

onException (exception SQLException)
    ConditionHandlingLib.HandleException(status, exception);
onException (exception AnyException)
    HandleOtherError(exception);
end
end

```

この例では、最初の **onException** ブロックで **SQLException** レコードが必要になり、2 番目のブロックで **AnyException** ステレオタイプが必要になります。この場合、EGL で SQL エラーが生じると、最初の **onException** ブロックが実行されます。別の種類のエラーであれば、2 番目のブロックが実行されます。**AnyException** ステレオタイプは、一般ユーザーの例外レコードのステレオタイプで、ステレオタイプに関係なく、どのようなエラーに応答する場合にも使用されます。

例外レコードにはすべて、最低 2 つのフィールドがあります。1 つはエラーのエラー・コードを示す **messageID** フィールドで、もう 1 つは問題の簡単な説明を示すメッセージ・フィールドです。ステレオタイプに応じて、例外レコードは他のフィールドを持つことがあります。例えば、**SQLRecord** ステレオタイプには、ステートメントのステータスについての **CHAR(5)** 値を保持する **sqlState** フィールドと DBMS からの **INT** 戻りコードを保持する **sqlCode** フィールドがあります。

無効な情報の受け渡し、存在しないレコードの削除または更新、あるいは、同じ主キーを持つレコードの追加を行って、例外処理を試してみることができます。次の例では、同じ ID 番号を持つ新規顧客レコードを、データベースに既存の行として追加することを試みます。この結果、主キーとの競合が発生し、EGL が **SQLException** 例外をスローします。

1. **programs** パッケージ内に、**exceptionHandlingTest** という名前の新規プログラムを作成します。
2. 新規プログラムの **main** 関数に、顧客レコード変数を作成し、データベース内に既に存在している ID 番号などの情報をそれに割り当てます。

```

package programs;

import eglDerby7.data.Customer;

program exceptionHandlingTest type BasicProgram {}

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

end

```

3. **try** ブロックで、データベースにレコードを追加します。

```

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

```

この **try** ブロックは、エラーに応答するための対応した **onException** ブロックがないため、あまり有用ではありません。

4. **try** ブロックを閉じる **end** の前に、**SQLException** を処理する **onException** ブロックを追加します。この例外は、EGL が **try** ブロックでコードの実行を試みる際に発生します。

```

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

```

```

    SysLib.writeStdout("message = "::ex.message);
    SysLib.writeStdout("SQLCode = "::ex.sqlCode);
    SysLib.writeStdout("SQLState = "::ex.sqlState);
end

```

完成したプログラムは次のようになります。

```

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. プログラムを保存し、生成して実行します。 コンソールに、**onException** ブロックからの出力が表示されます。

```

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

```

データベースへの変更のロールバック

ほとんどのリレーショナル・データベースは、Derby も含め、リカバリー可能リソースです。リカバリー可能リソースに変更を行う場合、コミットを発行し、その変更を保存してはじめて変更が永続的なものになります。例外が生じた場合も、このように、それらの変更をコミットする前にデータベースへの変更を元に戻すことができます。

sqlCommitControl ビルド記述子オプションを **NOAUTOCOMMIT** に設定することができます。これにより、変更が行われるたびに **EGL** がそれを自動的にコミットすることがなくなります。自動コミットをオフにした場合、手動で **sysLib.commit()** を呼び出して変更をコミットし、**sysLib.rollback()** を呼び出して、最後のコミットから行われた変更を元に戻すことができます。 **sysLib.commit()** で変更をコミットしないと、メインプログラムの終わりなどの実行単位の終わりに、変更が必ずコミットされます。

例えば、新規顧客を追加し、顧客からの最初のオーダーに関するオーダー情報を入力するなど、データベースにいくつかの変更を行っているとした場合。これらの操作の一部が成功した後で別の部分が失敗した場合

は、顧客が入力されてオーダーが入力されなかったり、オーダーが入力されて顧客が入力されなかったりすることがあります。この場合、次のようにして、ロールバックを **onException** ブロックに含めて、変更を元に戻し、データベースに不完全な情報が含まれないようにします。

1. **sqlCommitControl** ビルド記述子オプションを NOAUTOCOMMIT に設定します。
 - a. 「プロジェクト・エクスプローラー」ビューで、EGLSQL プロジェクトのビルド記述子をダブルクリックして EGL ビルド・パーツ・エディターで開きます。このファイルは、EGLSQL/EGLSource/EGLSQL.eglbid にあります。
 - b. ビルド・パーツ・エディターで、「指定したオプションのみを表示」チェック・ボックスをクリアします。
 - c. 「オプション」から、**sqlCommitControl** を見つけます。
 - d. **sqlCommitControl** の値を NOAUTOCOMMIT に設定します。

注: 編集するために **sqlCommitControl** オプションを開くには、横にある「値」列で 2 回ゆっくりとクリックします。「値」列で、3 回素早くクリックしてもかまいません。

- e. ビルド記述子を保管して、閉じます。
2. 前のセクションで作成した `exceptionHandlingTest` プログラムを開きます。
3. プログラムに既にある顧客レコードにオーダー・レコードを追加します。

```
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. **try** ブロック内で、最初にオーダーを、次に顧客を追加します。

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

この場合、オーダーは追加されますが、顧客は主キーが重複するため、追加されません。したがって、**onException** ブロック内でロールバックを呼び出すことで、ORDERS テーブルへの変更を元に戻すことができます。

5. **try** ブロックの後に、ロールバックを含めた **onException** ブロックを追加します。

```
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

```

これで、どちらかの **add** ステートメントが失敗した場合、コード **onException** ブロックが実行されて、データベースへの変更を元に戻そうとします。

6. **onException** ブロックの後に、例外が生じる前に追加されたオーダーがロールバックで正常に除去されたかをテストするコードを追加します。 オーダー・レコードがまだデータベース内にあるか確認するために、それを **noRecordFound** と比較できます。

```

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

```

データベースへの呼び出し状況を調べる別の方法は、**sqlLib.sqlData** システム変数の値をテストすることです。このシステム変数は 1 つのレコードであり、最後の SQL データベース操作が成功したか失敗したかを示す情報がそのフィールドに保管されます。具体的に言うと、フィールド

sqlLib.sqlData.sqlcode には、操作が正常に完了し、結果が返った場合は 0、操作は正常に完了したが、SELECT ステートメントによる結果が帰らなかった場合は 100 が含まれます。したがって、次のコードにより、行が検出されたかどうかもわかります。

```

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. プログラムを生成し、実行します。

プログラムの出力に、オーダーがデータベースに追加されていること、例外が発生していること、さらに、ロールバックの結果としてオーダーが正常に削除されたことが表示されます。

```

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.

```

これで、**exceptionHandlingTest.egl** ファイルのコードが完成しました。このファイル内にエラーがある場合 (赤の X 記号でマークされている場合) は、65 ページの『演習 7 で完成した **exceptionHandlingTest.egl** ファイル』に記載されているファイルのコードと、作成したコードが一致していることを確認してください。

演習のチェックポイント

この演習では、SQL 操作で生じたエラーの処理について基本的なことを学習しました。問題を予測し、アプリケーションをリカバリーできるよう例外処理を提供することは非常に重要です。

例外の原因となり得る実行時エラーについて詳しくは、「EGL Java ランタイムのエラー・コード」を参照してください。EGL で作成できる例外レコードのリストについては、「EGL コア例外レコード」を参照してください。

要約

この演習で学んだ内容

このチュートリアルを終了すると、次の概念およびタスクについての学習が完了します。

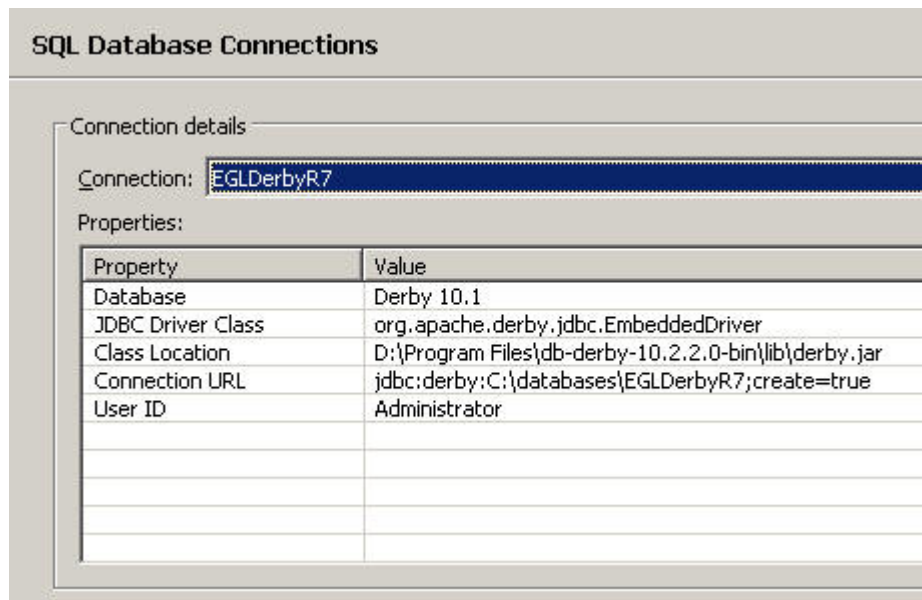
- **get** および **add** などのステートメントを使用した EGL でのデータベース処理の基礎
- 明示的な SQL 内、SQL ビルダー内、SQLRecord パーツのデフォルトで選択される条件内、準備済みステートメント内、あるいは **execute** ステートメントを使用して、独自の SQL ステートメントを作成する方法
- テーブル結合を実行して、より複雑なデータを処理する方法
- 明示的な SQL および準備済みステートメント内のホスト変数
- データ・アクセス・アプリケーションのエラーを処理する方法

トラブルシューティング

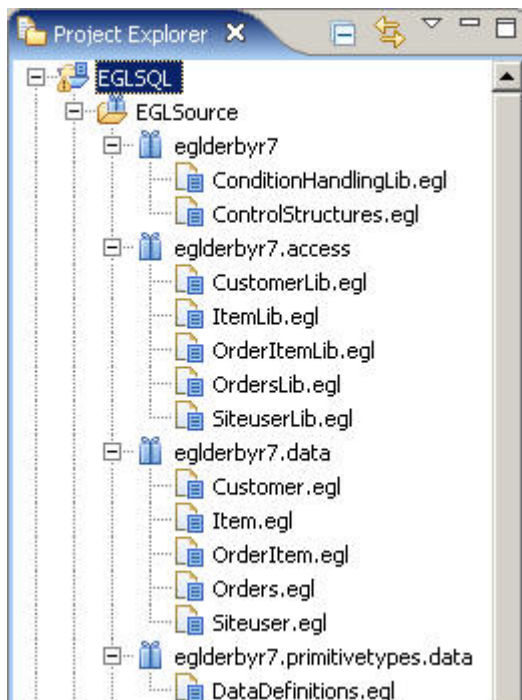
ここでは、チュートリアルアプリケーションで作業しているときに発生する可能性のある問題を挙げます。期待している結果が得られなかった場合に、ここに記載されている方法で問題を解決できることがあります。

- EGL コードに妥当性検査エラーまたは生成エラーがないか確認します。これらのエラーには、コード・エディターでは赤の X のアイコン、「EGL 生成結果」ビューではエラーでマークが付けられています。「EGL 生成結果」ビューにソース・ファイルがリストされ、緑のチェック・マークではなく赤の X のアイコンがある場合、ソース・ファイルは正常に生成されていません。このチュートリアルで使用されるほとんどの EGL コードは、資料の巻末に記載されています。55 ページの『リソース』を参照してください。
- すべてのファイルが保存済みであることを確認したら、プロジェクトを右クリックし、「生成」をクリックして、プロジェクト全体を生成します。
- 次を含めて、2 ページの『演習 1: データベース接続のセットアップ』のステップに完全に従っているか確認します。
 - プロジェクトのビルド記述子を開き、「**接続を使用した DB オプションのロード (Load DB options using Connection)**」リストで使用するデータベース接続を選択することにより、すべてのビルド記述子オプションを設定します。ビルド記述子に変更を加えた場合は、その後にプロジェクトを保存し、生成します。
 - Derby JAR ファイルをプロジェクトの Java ビルド・パスに追加します。
- 次のようにして、データベース接続が正しいことを確認します。
 1. 「ウィンドウ」 → 「設定」とクリックします。
 2. 「設定」ウィンドウで、「EGL」を展開して、「SQL データベース接続」をクリックします。
 3. 「接続」リストで、使用する接続を選択します。サンプル・データベースへの接続をまだ作成していない場合、通常、名前は EGLDerbyR7 ですが、(チュートリアル「EGL の紹介 (クイック・スタート・ガイド)」などで) 作成済みの場合は EGLDerbyR71 になります。
 4. 接続のプロパティが、ご使用のコンピューター上の関連ファイルの場所と一致していることを確認します。接続は次のようになります。ただし、derby.jar ファイルおよび EGLDerbyR7 フォルダー

(データベースが含まれています) は、ユーザーが指定した場所に置かれます。



5. プロパティがファイルの実際のある場所と一致しない場合は、「編集」をクリックし、設定を修正し、次に「テスト接続」をクリックして接続が機能することを確認します。
- 次のようにして、データ・パーツおよびロジック・パーツが正しく作成されていることを確認します。
 - このピクチャーのように、すべてのファイルが eglderbyr7 パッケージにあることを確認します。



- 例として、eglderbyr7.data パッケージ内の Customer.egl ファイルを開き、顧客レコードのパーツと次のレコードを比較します。

```
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

```

ご使用のパーツに誤りがある場合は、「EGL データ・アクセス・アプリケーション」ウィザードを再度ステップスルーし、パーツを再作成して誤りのあるものを書き直します。例えば、ご使用のレコード・パーツの **tableNames** プロパティーおよびそれらのレコード・パーツ内のフィールドの **column** プロパティーにあるそれぞれのテーブル名に EGL. 接頭部が含まれていない場合は、「テーブル名をスキーマで修飾」チェック・ボックスを選択し忘れたことを意味します。

実行時のエラー・メッセージ

以下は、このチュートリアルでプログラムの実行を試みているときに「コンソール」ビューに表示される可能性のあるいくつかのエラー・メッセージのリストです。

EGL0505E jdbc:derby:C:¥databases¥EGLDerbyR7;create=true に接続できません。データベース C:¥databases¥EGLDerbyR7 を開始するのに失敗しました。詳しくは、次の例外を参照してください。
(Cannot connect to jdbc:derby:C:¥databases¥EGLDerbyR7;create=true: Failed to start database 'C:¥databases¥EGLDerbyR7', see the next exception for details.)

データベースに基づいてパーツを作成するデータ・ツールがまだデータベースと接続されているため、EGL プログラムが接続できません。8 ページの『テスト・プログラムの実行』の説明どおりに、「データベース・エクスプローラー」ビューでデータ・ツール接続を閉じ、EGL プログラムを再実行を試みてください。

EGL0507E JDBC ドライバーのロード中にエラーが発生しました。(An error occurred while loading the JDBC drivers.)エラー: egl.core.RuntimeException: (Error: egl.core.RuntimeException:) EGL0009E vgj.jdbc.drivers プロパティーの値は必須です。(A value for the vgj.jdbc.drivers property is required.)

ビルド記述子が正しく設定されていません。ビルド記述子を開き、「接続を使用した DB オプションのロード (Load DB options using Connection)」リストで接続を選択して、ビルド記述子オプションの値をロードしてください。

EGL0507E JDBC ドライバーのロード中にエラーが発生しました。(An error occurred while loading the JDBC drivers.)エラー: java.lang.ClassNotFoundException: org.apache.derby.jdbc.EmbeddedDriver (Error:

java.lang.ClassNotFoundException: org.apache.derby.jdbc.EmbeddedDriver)

EGL が derby.jar ファイルを検出できません。4 ページの『データベースへのインポートおよび接続』の説明どおりに、このファイルをプロジェクトの Java ビルド・パスに追加したか確認してください。

リソース

このチュートリアルでは、以下のリソースを使用しました。

- 次のロケーションにある、サンプルの Derby データベース

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

shared_resources

ご使用の製品用の共用リソース・ディレクトリーで、例えば、Windows システムの場合は `C:\Program Files\IBM\SDP70Shared`、Linux システムの場合は `/opt/IBM/SDP70Shared` です。現行製品をインストールする以前より、EGL を含む IBM 製品の前のバージョンをインストールし、保持している場合は、前のインストールでセットアップされた共用リソース・ディレクトリーを指定する必要があります。

version

インストールされているプラグインのバージョン。これには、ピリオドで区切られた 3 つの数字、ストリング区切り文字、およびプラグインがビルドされた日時が含まれます。例:

7.0.0.RFB_20070120_1300。複数存在している場合は、古いバージョンを使用する理由がない限り、最も新しいバージョン番号のプラグインを使用してください。

- 56 ページの『演習 2 で完成した CRUDTest.egl ファイル』
- 57 ページの『演習 3 で完成した selectItems.egl ファイル』
- 58 ページの『演習 4A で完成した selectItems.egl ファイル』
- 60 ページの『演習 4B で完成した selectItems.egl ファイル』
- 62 ページの『演習 5 で完成した printCustomerOrders.egl および records.egl ファイル』
- 64 ページの『演習 6 で完成した CustomerTableOperations.egl および duplicateTable.egl ファイル』
- 65 ページの『演習 7 で完成した exceptionHandlingTest.egl ファイル』

以下に、このチュートリアルで扱うトピックでのヘルプ・システムへのリンクを示します。

- #sql ディレクティブ
- SQL データの処理
- SQLRecord ステレオタイプ
- SQL での add についての考慮事項
- SQL での delete についての考慮事項
- SQL での execute についての考慮事項
- SQL での forEach についての考慮事項
- SQL での get についての考慮事項
- SQL での open についての考慮事項
- SQL での prepare についての考慮事項
- SQL での replace についての考慮事項
- EGL Java ランタイムのエラー・コード
- EGL コア例外レコード

- EGL ライブラリー sqlLib
- commit()
- rollback()
- SQL ステートメントの処理

演習 2 で完成した CRUDTest.egl ファイル

このコードは、完成したバージョンの CRUDTest.egl ファイルです。ファイル内に赤い X 印がマークされたエラーが表示される場合は、ご使用のコードが次のコードと一致するか確認してください。

```
package programs;

import eglderby7.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

    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

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

```

10 ページの『演習 2: EGL での SQL の基本操作』に戻る。

演習 3 で完成した selectItems.egl ファイル

次のコードは、演習 3 で完成したバージョンの selectItems.egl ファイルです。このファイル内にエラーがある場合 (赤の X 記号でマークされます) は、作成したコードがこのコードと一致していることを確認してください。

```

package programs;

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

program selectItems type BasicProgram {}

function main()
    try
        printExpensiveItems(200);
    end
end

```

```

        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

```

18 ページの『演習 3: **open** および **forEach** を使用したカーソルの管理』に戻る。

演習 4A で完成した selectItems.egl ファイル

次のコードは、演習 4A で完成したバージョンの selectItems.egl ファイルです。ファイル内に赤い X 印がマークされたエラーが表示される場合は、ご使用のコードが次のコードと一致するか確認してください。

```

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

```

25 ページの『演習 4A: 準備済みステートメントの使用』に戻る。

演習 4B で完成した selectItems.egl ファイル

次のコードは、演習 4B で完成したバージョンの selectItems.egl ファイルです。ファイル内に赤い X 印がマークされたエラーが表示される場合は、ご使用のコードが次のコードと一致するか確認してください。

```

package programs;

import eglderby7.data.Item;
import eglderby7.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

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

```

28 ページの『演習 4B: **prepare** ステートメント内のホスト変数』に戻る。

演習 5 で完成した printCustomerOrders.egl および records.egl ファイル

次のコードは、演習 5 で完成したバージョンの printCustomerOrders.egl ファイルおよび records.egl ファイルです。このファイルのいずれかにエラーがある場合 (赤の X 記号でマークされます) は、作成したコードがこのコードと一致していることを確認してください。

```

[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

```

32 ページの『演習 5: テーブル結合を使用した、より複雑なデータの取得』に戻る。

演習 6 で完成した CustomerTableOperations.egl および duplicateTable.egl ファイル

次のコードは、演習 6 で完成したバージョンの CustomerTableOperations.egl ファイルおよび duplicateTable.egl ファイルです。このファイルのいずれかにエラーがある場合 (赤の X 記号でマークされます) は、作成したコードがこのコードと一致していることを確認してください。

```
[libraries/CustomerTableOperations.egl]

package libraries;

import eglDerby7.ConditionHandlingLib;
import eglDerby7.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 eglderby7.StatusRec;
import libraries.CustomerTableOperations;

program duplicateTable type BasicProgram {}

    status StatusRec;

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

end

```

36 ページの『演習 6: 任意の SQL コードの実行』に戻る。

演習 7 で完成した exceptionHandlingTest.egl ファイル

次のコードは、演習 7 で完成したバージョンの exceptionHandlingTest.egl ファイルです。このファイル内にエラーがある場合 (赤の X 記号でマークされます) は、作成したコードがこのコードと一致していることを確認してください。

```

package programs;

import eglderby7.data.Customer;
import eglderby7.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);
            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
```

47 ページの『演習 7: 例外処理およびロールバック』に戻る。

付録. 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-8711
京都港区六本木 3-2-12
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態を提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

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

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性があります、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があります、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生した創作物にも、次のように、著作権表示を入れていただく必要があります。

© (C) (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。© Copyright IBM Corp. _年を入れる_. All rights reserved.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

商標

以下は、IBM Corporation の商標です。

IBM
Rational

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。



Printed in Japan