Chapter 11

DATA ACCESS WITH SQLJ—EMBEDDING SQL IN JAVA

This chapter covers the use of embedded SQL in Java, known as SQLJ technology. The focus of the chapter is on how to use SQLJ rather than JDBC to perform database operations from a Java application. The topics covered include:

- Connecting to the database
- Executing SQL statements
- Processing SQL object types
- Processing SQL collections

The chapter also discusses Oracle JPublisher in more detail and shows how you can use the Java classes generated by JPublisher when processing object types or collections in SQLJ.

> **Note**
>
> In order to focus on the key functionality discussed, and to minimize the amount of extra code written and documented, the examples shown in this chapter do not use proper code-management techniques for closing connections and error handling. For example, the code listings close database connections in the try section of a try-catch block. This does not cater for the case when an error occurs. It is better practice to close a database connection in the final section of a try-catch-finally block to ensure that the connection is closed for the success and failure conditions in your code.

11.1 AN OVERVIEW OF SQLJ

For years, vendors have provided a way for third-generation languages to execute SQL, in order to access data in relational databases. It was only natural that the same would happen with Java. The major benefits to developers are:

- Faster development because SQLJ typically requires fewer lines of code when compared to using JDBC.
- Early validation of SQL syntax at compile time, leading to more robust code.

SQLJ allows you to embed static SQL statements in Java code in a way that is compatible with the Java design principles. Static SQL statements specify predefined operations that do not change at runtime. Dynamic SQL statements specify SQL operations that are not predefined at compile time, where as a Java program can construct the SQL statement on the fly at runtime.

SQLJ and JDBC code can be mixed in the same program, and, as well, can share connections and structures.

11.1.1 SQLJ COMPONENTS

SQLJ consists of two primary components:

1. An SQLJ translator
2. An SQLJ runtime

One other component is a customizer that can be used to tailor SQLJ profiles for a specific database.[1] Since Oracle SQLJ Translator uses an Oracle-supplied customizer, you can access the extended features of an Oracle RDBMS environment.

11.1.1.1 The SQLJ Translator. The SQLJ translator is a preprocessor that reads SQLJ source code from a file with an `.sqlj` extension,[2] and produces a Java source in a `.java` file and one or more SQLJ profile files.[3] The SQLJ translator automatically compiles the Java source to produce the class file.

The Oracle SQLJ translator is available as a command-line utility for manually translating an SQLJ file into the Java source and class. Oracle JDeveloper automatically invokes the SQLJ translator when you build an SQLJ file. During the translation of the SQLJ source to the Java source, you can add an SQLJ translator command-line option to perform syntax checking of embedded SQL statements.

The SQLJ Translator class library is located in a file called translator.zip, which can be found in the `ORACLE_HOME/sqlj/lib` sub-directory of your Oracle8i installation. The `translator.zip` file and the SQLJ runtime classes must be in the class path at development time.

11.1.1.2 The SQLJ Runtime. SQLJ Runtime is automatically invoked when a Java program containing SQLJ code is executed. The SQLJ Runtime component implements the SQL operations embedded in the SQLJ code. Oracle SQLJ runtime requires the use of an Oracle JDBC driver to access the database, even though the SQLJ standard does not require the SQLJ runtime to use a JDBC driver to access a database. The SQLJ Runtime component class library is found in the file `ORACLE_HOME/sqlj/lib/runtime.zip`. This file must be in the class

---

[1] An SQLJ profile is a class, usually in serialized form, used by the SQLJ runtime to invoke SQL operations.

[2] The Oracle SQLJ Translator is written in pure Java.

[3] An SQLJ profile file contains serialized Java objects in a file with a `.ser` extension. Alternatively, a SQLJ profile can be created in a `.class` file. The serialized Java objects contain details about the embedded SQL operations in your SQLJ source code.

path when you are executing an SQLJ application. Note that the SQLJ Translator is not required at runtime.

11.1.2 CREATING AN SQLJ FILE

Any text editor or development environment with an editor can be used to create an SQLJ file. Oracle JDeveloper is a development environment that provides support for creating SQLJ files. The key features of an SQLJ file are:

- The file must have an extension named **`sqlj.`**
- The file must contain Java code for a Java class definition, with or without embedded SQLJ statements. The file name must be the same as the name of the public Java class contained in the file.
- The file must import the following two packages for compilation of SQLJ runtime classes that are generated or used:
  - sqlj.runtime.*
  - sqlj.runtime.ref.*

Listing 11.1 shows the basic structure of an SQLJ source, and the general syntax used to embed an SQL statement in the Java code.

File name: **`RegisterCustomer.sqlj`**

```
01: import sqlj.runtime.*;
02: import sqlj.runtime.ref.*;
03:
04: public class RegisterCustomer {
05:
06:   public static void main(String[] args) {
07:     #sql { sql-statement };
08:   }
09: }
```

LISTING 11.1 SQLJ source with generic embedded SQL statement

Listing 11.1 notes:

- Lines 1 and 2 are the import statements for the required classes found in SQLJ packages.
- Line 4 is the Java class name, which is the same as the file name and contains the methods that executes SQLJ statements.

- Line 7 is an example of the generic syntax for an SQL statement when it is embedded in your Java source code.
  - The text **#sql** must precede all SQLJ statements.
  - All SQL statements are placed between braces with no semicolon terminator inside the braces. A semicolon is placed outside the closing brace of the SQLJ statement.

Additional imports may be required, depending on the Java class used by the SQLJ source code. The SQLJ code is still primarily a Java source file.

11.1.3 TRANSLATING THE SQLJ FILE

The SQLJ translator converts the SQLJ source into a Java source that is automatically compiled into a class file.

11.1.3.1 Running the SQLJ Translator. The steps in running the SQLJ translator from the command line are:

1. Add the file ORACLE_HOME/sqlj/lib/translator.zip to your class path.
2. Run the **sqlj** command-line utility. The SQLJ translator executable is located in your ORACLE_HOME/bin directory.[4]

The steps are shown in Listing 11.2.

```
Step 1: Set the Classpath
For Windows NT:
  set CLASSPATH=D:\orant8i\sqlj\lib\translator.zip;%CLASSPATH%

For Unix Bourne or Korn Shell:
  CLASSPATH=$ORACLE_HOME/sqlj/lib/translator.zip:$CLASSPATH
  export CLASSPATH

For Unix C-Shell:
  setenv CLASSPATH $ORACLE_HOME/sqlj/lib/translator.zip:$CLASSPATH

Step 2: Run the SQLJ Translator
sqlj [options] file.sqlj
```

LISTING 11.2 Using the SQLJ translator

[4]If you are using Oracle JDeveloper, the SQLJ Translator command-line utility is found in the `bin` subdirectory, relative to your base directory for the JDeveloper installation. This will typically be `<drive>:\Program Files\Oracle JDeveloper 3.0\bin`, if you are using JDeveloper 3.0.

If you enter the SQLJ command by itself, the application prints a brief listing showing some help text for the command-line syntax and options.

The SQLJ translator has several command-line options. The command options are entered as:

```
-name
```

or,

```
-name=value
```

The hyphen, as shown, must immediately precede the option name. The option's value is either true or false. Turn on the option by entering the option as-name or –name=true, otherwise you must enter –name=false to turn the option off. Table 11.1 is a brief list of SQLJ translator command-line options.

TABLE 11.1 SQLJ translator command-line options

OPTION	DESCRIPTION AND VALUES
-user=*user/password*	The username and password used by the SQLJ translator to log into a database specified by the *url* option. The `user` option is only used if you want the SQLJ translator to perform syntax checking of embedded SQL statements.
-url=*jdbc-url*	The JDBC URL that identifies the database used for validating SQL statements and the database structures on which they operate. By default the URL is "`jdbc:oracle:oci8:@`".
-d=*directory*	Specifies the output root directory for generated binary (`ser` and `class`) files. The generated Java source file is not affected by this option.
-status	Displays status messages to the screen during the translation process.
-compile=false	Suppresses compilation of the generated Java source. No class files are created. Compilation is performed by default.
-ser2class	Creates profile files in the form of `.class` files, not `.ser` files. *Note:* This option should be used if your JVM environment does not support loading SER files.
-J-*option*	Specifies command-line options for the JVM that runs the SQLJ translator.
-classpath=*classpath*	Specifies the CLASSPATH to the JVM (java) and compiler (javac) used by the SQLJ translator.
-linemap	Causes the translator to generate SQLJ source-line numbers as comments in the Java source.

You specify command-line options separated by one or more spaces. For example:

```
sqlj –user=bookstore/bookstore -linemap
     –url=jdbc:oracle:thin:@localhost:1521:ORA815 file.sqlj
```

The example should be entered on one line. For clarity, the example is shown on more than one line with no command-line continuation characters, which are platform-dependent, if supported.

The SQLJ translator performs the following steps in sequence depending on the options used:

1. The Java Virtual Machine invokes the SQLJ Translator.
2. The translator parses the SQLJ source code, checking for proper SQLJ syntax.
3. The semantics checker is invoked to check whether the embedded SQL statements use valid database structures, such as columns, tables, procedures, data type validation, and more.
4. The SQLJ source code is converted into a Java source that makes calls to the SQLJ runtime API. One or more SQLJ profiles are also created (see below, "Profile Files," for more information). The SQLJ translator also generates a file known as the *profile-keys class*, which is the class definition file for a specialized class used in conjunction with the profiles. The profile-keys class is used to load and access serialized profiles, and contains mapping information between the SQLJ runtime calls and their SQL operations stored in a serialized profile. The SQLJ Runtime is called to implement the actions of your embedded SQL operations.
5. Normally, the SQLJ Translator invokes the Java compiler to compile the generated Java source, and, optionally, produce a class file for each of the serialized resource files (`.ser` files) if you specified the `–ser2class` option.
6. The Oracle SQLJ customizer is invoked. This step can be suppressed if you use the option: `-profile=false`.

11.1.3.2 Files Generated by the SQLJ Translator. If your SQLJ source file name is called `ShoppingCart.sqlj`, then the SQLJ Translator generates at least the following files:

1. `ShoppingCart.java`—the generated Java source, which includes calls to the SQLJ Runtime to implement the operations specified by SQLJ statements.
2. `ShoppingCart.class`—the compiled version of the generated Java source.

3. `ShoppingCart_SJProfileKeys.ser`—contains mappings for the SQLJ Runtime calls in your application and the SQL operations stored in the serialized profile.
4. `ShoppingCart_SJProfile0.ser`—is the generated profile file describing the SQL operations to be performed.

Profile Files

The generated profile files contain information about all of the embedded SQL statements in your SQLJ source code. This includes:

1. SQL operations to execute.
2. Tables to access.
3. Stored procedures and functions to call.
4. The data types being manipulated.

The SQLJ Runtime accesses the profile files (using information in the profile-key class) to retrieve the SQL operations and pass them to the JDBC driver for processing.

By default, profiles are placed in serialized resource files, each with a `.ser` extension. The SQLJ Translator command-line option `–ser2class` is used to specify that the profiles should be created as a `.class` file.

SQL operations are executed in the context of a database connection called a *connection context*. The SQLJ language provides a way to create more than one connection context in your application. This would be typical for applications needing to manage data in different databases, such as a funds-transfer application. The SQLJ Translator generates a profile file for each connection context used in your application A unique number, starting from 0 and incremented by 1 for each connection context, is appended to the file name.

11.1.4 RUNNING THE SQLJ FILE

In general, any Java program that connects to a database can be SQLJ enabled. For example, if the SQLJ file contains a `main` method, as used in a standard Java application, you can run the class generated by the SQLJ Translator using the `java` command-line tool. If you created an SQLJ source as a Java applet, you can run the Applet class generated by the SQLJ Translator in a Java-enabled Web browser.

Oracle JDeveloper has built-in support for compiling, debugging, and executing SQLJ applications. Oracle JDeveloper debugger makes it easier to debug SQLJ code by allowing you to debug from the SQLJ source rather than the generated Java source.

11.2 CONNECTING TO A DATABASE IN SQLJ

An SQLJ application always has a connection context. Normally, if your application is only using a single database connection, you use the default connection context. If you require additional database connections from the same application, you can create a named connection context.

11.2.1 SETTING THE DEFAULT CONNECTION CONTEXT

Creating a default connection does not require an explicit context name. The process has two main steps:

1. Load a database JDBC driver class or register an instance of the driver.
2. Set the default connection context by providing a connection string and optional username/password.

In order to set the default connection context, you need to create a DefaultContext object. The DefaultContext object is typically used for applications that require a single database connection.

Listing 11.3 shows two ways to create a connection using a DefaultContext object to set the default connection context for SQLJ statements in your application.

```
01: package com.prenhall.OFJP.sqlj;
02: import sqlj.runtime.*;
03: import sqlj.runtime.ref.*;
04: import java.sql.*;
05:
06: public class MakeConnection {
07:   public MakeConnection(String url) {
08:     try {
09:       DriverManager.registerDriver(
10:          new oracle.jdbc.driver.OracleDriver());
11:
12:       // Setting a Default Context from a JDBC connection
13:       Connection conn = DriverManager.getConnection(url);
14:       DefaultContext dctx = new DefaultContext(conn);
15:       DefaultContext.setDefaultContext(dctx);
16:
17:       String result;
18:       #sql { select user||' '||to_char(sysdate)
```

LISTING 11.3 Creating a connection for the default context

```
19:              into :result
20:              from dual };
21:       System.out.println("result: " + result);
22:
23:       dctx.close();
24:     }
25:     catch (Exception e) {
26:       e.printStackTrace();
27:     }
28:   }
29:
30:   public static void main(String[] args) {
31:     new MakeConnection("jdbc:oracle:thin:" +
32:        "bookstore/bookstore@localhost:1521:ORA815");
33:   }
34: }
```

LISTING 11.3 *Continued*

Notes for Listing 11.3:

- Lines 2 and 3 are required imports for an SQLJ application. The java.sql package is present because JDBC calls are used to load the JDBC driver and create a JDBC Connection.
- Lines 9–13 load the JDBC driver and create a JDBC connection using the DriverManager.
- Line 14 creates a DefaultContext object to be associated with the JDBC Connection created in line 13.
- Line 15 sets the default context for all unqualified SQLJ statements, which execute using the underlining JDBC connection. This example shows how you can work with SQLJ and JDBC code in the same application.
- Lines 18–20 show an SQLJ statement executing a SELECT statement to query the current database user name and the current date.
- Line 23 closes the default connection context. Note that the JDBC connection is also closed by this action. You can keep the underlying JDBC connection open by calling the DefaultContext close() method with a `false` value for the boolean argument. For example:

  ```
  dctx.close(false)
  ```
  [5]

[5]You can use the class constant `ConnectionContext.KEEP_CONNECTION` as the parameter value, instead of the keyword `false`.

Since all SQL code embedded as SQLJ statements can throw an SQLException, the code should either be enclosed in a try-catch-finally block or the exception must be propagated to the caller.

11.2.1.1 Alternative Ways of Setting the Default Context. An alternative way to set the default context is:

```
01: Class.forName("oracle.jdbc.driver.OracleDriver");
02: DefaultContext.setDefaultContext(new DefaultContext(url, false));
```

Line 1 loads the JDBC driver, which is required for line 2 to work.

Line 2 sets the default connection context for SQLJ statements that have not been qualified with a connection context name. The DefaultContext object is used to create the connection context used by the setDefaultContext() method.

There are more than one DefaultContext constructors. The example uses a form of the constructor that has a URL as the first parameter, and a boolean value as the second parameter to set the auto-commit state to false for the underlying JDBC connection.

If you want control over transaction processing, set the auto-commit mode to false. Another way to set auto-commit to false is to use the connection object associated with the default context. For example:

```
// Assume you have used the DefaultContext constructor with a
// connection object
DefaultContext dctx = new DefaultContext(
                                DriverManager.getConnection(url));

// Set the connection associated with the default context to false
dctx.getConnection().setAutoCommit(false);
```

Another Oracle-specific way to create a default context is to use the static `connect()` method in the `oracle.sqlj.runtime.Oracle` class. For example:

```
DefaultContext dctx = Oracle.connect(url);
```

Using the `Oracle.connect()` method performs the following operations:

- Loads the Oracle JDBC driver.
- Connects to the database specified by the URL parameter.
- Sets the default context and returns the default context.

If you use either of the alternative techniques discussed here, then you do not need to import the java.sql package. However, if you use the `Oracle.connect()` technique, your code will only work with the Oracle SQLJ class libraries, and so is less portable.

Once you have a default context object, you can obtain the associated JDBC connection as follows:

```
Connection conn = dctx.getConnection();
```

Regardless of which method you use, you can still mix JDBC calls and SQLJ statements in the same application.

11.2.2 CREATING AND USING ADDITIONAL CONNECTION CONTEXTS

Your SQLJ application may need more than one database connection. This is achieved by creating additional connection contexts. Each connection context is associated with a new database connection, either to the same or a different database. To create an additional connection context:

- Declare a connection context class, using an SQLJ context declaration.

  For example:

  ```
  #sql context MyContext;
  ```

In this example, the SQLJ Translator generates a class called MyContext. The SQLJ language specification provides for declarations and statements. There are two types of declarations:

1. Context class declarations (discussed here)
2. Iterator class declarations (discussed below in section 11.3.2, subsection "Reading Multiple Rows Using Iterators")

The SQLJ Translator generates a class with a name as specified by you in the declaration. A declaration type can be preceded by Java modifiers, such as `public`, `private`, or `protected`, and followed by an `implements` clause, as follows:

```
#sql <modifiers> context ClassName implements InterfaceName, …;
```

To use the connection context:

1. Instantiate an object for the new connection context class.

2. Qualify the SQLJ statements using the object variable name, enclosed in square brackets,[6] for the new connection context object.

   For example:

```
01: package com.prenhall.OFJP.sqlj;
02: import sqlj.runtime.*;
03: import sqlj.runtime.ref.*;
04: import oracle.sqlj.runtime.*;
05:
06: #sql context MyContext;
07:
08: public class NewContext {
09:   public NewContext(String url1, String url2) {
10:     String userName1, userName2;
11:     try {
12:       Class.forName("oracle.jdbc.driver.OracleDriver");
13:       DefaultContext.setDefaultContext(
14:           new DefaultContext(url1, false));
15:       // Execute in the default context
16:       #sql { select user into :userName1 from dual };
17:       System.out.println("User(default context): " + userName1);
18:
19:       MyContext ctx = new MyContext(url2, false);
20:       // execute in the additional context
21:       #sql [ctx] { select user into :userName2 from dual };
22:       System.out.println("User(new context): " + userName2);
23:
24:       DefaultContext.getDefaultContext().close();
25:       ctx.close();
26:     }
27:     catch (Exception e) {
28:       e.printStackTrace();
29:     }
30:   }
31:
32:   public static void main(String[] args) {
33:     new NewContext(
34:       "jdbc:oracle:oci8:bookstore/bookstore@ORA815",
35:       "jdbc:oracle:oci8:scott/tiger@ORA815");
36:   }
37: }
```

LISTING 11.4 Using more than one connection context

[6] In the SQLJ syntax, square brackets are required around the object reference name to qualify statements executed in the specified context.

Notes for Listing 11.4:

- In line 6, the SQLJ context declaration creates the context class MyContext to be used for creating additional connection context objects.
- Lines 13 and 14 set the default context using the connection formed from the URL in url1.
- Line 16 executes a SQL SELECT statement using the default context.
- Line 19 creates a new context object using the MyContext class, for the URL in url2.
- Line 21 executes another SQL SELECT using the connection context `ctx` defined as a MyContext object. To execute the SQLJ statement in the second connection context (MyContext), you use the `ctx` reference variable in square brackets, i.e., **`[ctx]`**, and place it between the `#sql` token and the SQL statement.
- Line 24 closes the default connection.
- Line 25 closes the new connection context.

The SQLJ Translator generates a `.java` file, and a `.class` file for the `MyContext` SQLJ declaration on line 6 for the `NewContext` class.

Listing 11.5 shows some of the Java code generated for the following SQLJ declaration:

```
#sql context MyContext;
```

```
01: class MyContext
02: extends sqlj.runtime.ref.ConnectionContextImpl
03: implements sqlj.runtime.ConnectionContext
04: {
05:   public MyContext(Connection conn) throws SQLException {
06:     super(profiles, conn);
07:   }
08:
09:   public MyContext(String url, String user,
10:                    String password, boolean autoCommit)
11:     throws SQLException {
12:     super(profiles, url, user, password, autoCommit);
13:   }
14:   :
15:   private static final sqlj.runtime.ref.ProfileGroup
16:               profiles = new sqlj.runtime.ref.ProfileGroup();
17: }
```

LISTING 11.5 Example of a generated context class

Notes for Listing 11.5:

- Line 2 shows that the connection context `MyContext` is a subclass of `ConnectionContextImpl` in the `sqlj.runtine.ref` package, and implements the `ConnectionContext` interface from the `sqlj.runtime` package.
- Lines 5–7 and 9–13 represent two of the five constructors that must be provided for a connection context class definition.
- The other methods produced by the SQLJ Translator, but not shown in the example, include:
  - The getDefaultContext() and setDefaultContext() methods for getting and setting the default context, respectively.
  - The getProfileKey() method for getting the profile key file.
  - The getProfile() method for getting a profile file used for the connection context.

It is not important to delve into the details of these classes unless you need to make customizations of your own. The rest of this chapter focuses on the usage of SQLJ technology, and occasionally shows some of the underlying code generated.

11.2.3 EXECUTION CONTEXTS

Each connection context is created with an implicit execution context object. The execution context provides the environment in which an SQL operation is executed.

The execution context class, called `sqlj.runtime.ExecutionContext`, contains accessor methods for execution control, status, and cancellation of an SQL statement. The execution control methods modify the semantics of subsequent SQL operations. The execution status methods describe the results of the last SQL operation. For example, they detect the number of rows modified by an UPDATE statement. The execution cancellation methods terminate the current SQL operation.

The code snippet that follows, in the MySQLJApp class, shows how you can obtain a reference to an `ExecutionContext` object that is implicitly created with the default connection context. The code uses the execution context to determine the number of rows affected by an SQL UPDATE statement.

```
01: package com.prenhall.OFJP.sqlj;
02: import sqlj.runtime.*;
03: import sqlj.runtime.ref.*;
04:
05: public class MySQLJApp {
```

```
06:   public static void main(String[] args) {
07:     String url = "jdbc:oracle:thin:" +
08:                  "bookstore/bookstore@localhost:1521:ORA815";
09:     try {
10:       if (args.length == 1) {
11:         url = args[0];
12:       }
13:       Class.forName("oracle.jdbc.driver.OracleDriver");
14:
15:       DefaultContext dCtx = new DefaultContext(url, false);
16:       DefaultContext.setDefaultContext(dCtx);
17:
18:       ExecutionContext exeCtx = dCtx.getExecutionContext();
19:
20:       #sql { update courier
21:                set cost_per_item = cost_per_item * 1.1 };
22:
23:       System.out.println(exeCtx.getUpdateCount() +
24:                          " row(s) updated");
25:       dCtx.close();
26:     }
27:     catch (Exception e) {
28:       e.printStackTrace();
29:     }
30:   }
31: }
```

The bold text in line 18 shows how to obtain the implicit execution context object associated with the default connection context. In line 23, the number of rows affected by the preceding SQL UPDATE operation is determined by calling the execution context getUpdateCount() method. The getUpdateCount() accessor method is referred to as a status method.

When writing a multithreaded application, you may want each thread to manage the execution control, status, and cancellation of its own SQL operations. You can create an execution context object for each thread that uses the same connection context. To use multiple execution contexts, you explicitly create the execution context object and qualify each SQL operation with the execution context variable. The execution context variable inside square brackets is placed between the #sql token and the SQL operation.

The next two code snippets show how to explicitly associate an execution context with an SQL statement. The first example uses the default connection context, and the second uses a named connection context.

11.2.3.1 ExecutionContext with a Default Connection Context.

```
ExecutionContext exeCtx = connCtx.getExecutionContext();

#sql [exeCtx] { update courier
                set cost_per_item = cost_per_item * 1.1 };
```

The default connection context is used because it is absent from the SQLJ statement.

11.2.3.2 ExecutionContext with a Named Connection Context. If you have multiple connection contexts and want to execute an SQL operation in an explicit execution context on a specific connection, use the following syntax in the SQLJ statement:

```
DefaultContext connCtx = new DefaultContext(url, false);
DefaultContext.setDefaultContext(connCtx);
ExecutionContext exeCtx = connCtx.getExecutionContext();

#sql [connCtx, exeCtx] { update courier
                set cost_per_item = cost_per_item * 1.1 };
```

This example shows that you place the connection context variable `connCtx`, followed by the execution context variable `exeCtx`, inside the square brackets and separated by a comma. The connection context variable must appear before the execution context variable.

The ExecutionContext class has accessor methods known as *status methods*:

- ❑ getWarnings()—to get the first warning for the most recent SQL statement. Then call getNextWarning() to access chained warnings.
- ❑ getUpdateCount()—the number of rows affected by the SQL statement.

It also has accessor methods known as *control methods*:

- ❑ setBatching(boolean), isBatching()—to set set batching or determine whether batching is in operation.
- ❑ setBatchSize(), getBatchSize()—to set or get the size of batching operations.
- ❑ setMaxRows(), getMaxRows()—to set or get the number of rows that can be processed for a query operation, excess rows are silently ignored.

The accessor methods known as *cancellation methods* are:

- ❑ setQueryTimeout(), getQueryTimeout()—to manage query execution time.
- ❑ cancel()—to abort a query when executing in a multithreaded environment.

11.3 EXECUTING SQL STATEMENTS USING SQLJ

The major benefit of executing SQL statements in SQLJ is the simplicity of accessing the database from a Java program, particularly if you are already conversant with the SQL language. If the SQL statement is a complex one, requiring or returning many column values, many lines of JDBC code can be reduced to one embedded SQLJ statement, excluding the declaration of variables required to store values for the SQL statement.

Consider the following SQL statement:

```
SELECT name, surname, email FROM customer WHERE id = value
```

If you want to process this statement with JDBC calls, the following fragment of code to read a single row is required (ignoring the establishment of the database connection):

```
01: public void getCustomerInfo(Connection conn, int id) {
02:   String name;
03:   String surname;
04:   String email;
05:
06:   PreparedStatement ps = conn.prepareStatement(
07:      "select name, surname, email from customer where id = ?");
08:   ps.setInt(1, id);
09:   ResultSet rs = ps.executeQuery();
10:   if (rs.next()) {
11:     name = rs.getString(1);
12:     surname = rs.getString(2);
13:     email = rs.getString(3);
14:     System.out.println(
15:       "Customer: " + name + " " + surname + " " + email);
16:   }
17:   rs.close();
18:   ps.close();
19: }
```

The equivalent code to read a single row in SQLJ is:

```
01: public void getCustomerInfo(int id)
02: throws SQLException {
03:   String name;
04:   String surname;
```

```
05:   String email;
06:
07:   #sql { select name, surname, email into :name, :surname, :email
08:          from customer where id = :id };
09:   System.out.println(
10:     "Customer: " + name + " " + surname + " " + email);
11: }
```

The SQLJ code requires less typing than the JDBC code, and is far easier for the programmer to read. In addition, using the SQLJ Translator command-line options, you can validate the SQL statement at compile time, which you cannot do with JDBC statements.

Line 7 of the SQLJ code example introduces some interesting syntactic elements of an SQLJ statement:

- Unlike with JDBC, no quotes are used around the SQL statement.
- In the SQL statement, you bind the values from, or into, Java host variables by using a colon immediately preceding the Java variable name.
- The SQL statement can be split over one or more lines, with no need for line-continuation characters.
- The SQL statement must be enclosed inside braces, and a semicolon is placed outside the closing brace.

The SQL SELECT statement in the SQLJ example is limited to fetching only one row, because the Java variables can only contain one variable at a time. In the SQLJ example, an SQLException is thrown if either of the following cases arise:

- No rows are returned by the query.
- More than one row is returned by the query.

Although these two conditions can occur in the JDBC code example, exceptions are not thrown. The JDBC code gives you more control over managing these error conditions.

If you wish to process more than one row in a SQLJ application, you need to use an SQLJ iterator (refer to section 11.3.2, subsection “Reading Multiple Rows Using Iterators”).

11.3.1 USING HOST VARIABLES

In an SQLJ statement, you use or modify the value of a Java variable by prefixing it with a colon. A Java variable prefixed with a colon in an SQLJ statement is known as a *bind/host variable*. For example:

```
int customerId = 10;
String customerName;

#sql { SELECT name INTO :customerName
       FROM customer
       WHERE id = :customerId  };
```

The SQLJ Translator sets the mode of a host variable as *input*, *output*, or *input-output*, depending on the context of the variable usage in the SQL statement. By default, the mode is IN, except when the host variable is part of an INTO list in a SELECT statement, or is the target of an assignment in an SQLJ SET statement, in which case the mode is OUT. You can explicitly set the mode of the host variable by using one of the following mode specifiers:

- IN—for input only.
- OUT—for output only.
- INOUT—for input and output.

Mode names are case-insensitive. For example:

```
:mode-specifier hostVar

where mode-specifier = IN or OUT or INOUT, for example:

:IN hostVar
:in hostVar

:OUT hostVar
:INOUT hostVar
```

For readability, it is recommended that no spaces appear between the colon and the mode specifier. At least one space character is required between the mode specifier and the host variable name.

You can create *host expressions* for input values by enclosing an expression in parentheses after the colon or mode specifier. Examples are:

```
:(hostVar1 + hostVar2)
:IN(hostVar1 * hostVar2)
```

The parentheses, as shown, are required to enclose the expression. To set the value of a Java host variable, you can uses a host expression with the SQLJ SET statement. For example:

```
java.sql.Date hostVar1;

#sql { SET :hostVar1 = to_char(sysdate) };

// This is equivalent to:

#sql { SET :OUT hostVar1 = to_char(sysdate) };

// This equivalent to an embedded PL/SQL block:

#sql { BEGIN :OUT hostVar1 := to_char(sysdate); END; };
```

> **Note**
>
> These PL/SQL block expressions are evaluated prior to the SQLJ statement being executed.

The rule for using host variables is: Prefix the Java variable with a colon if it is inside the braces of the SQLJ statement.

11.3.2 USING DML AND DDL STATEMENTS IN SQLJ

Executing an SQL data manipulation or data definition language statement is as simple as placing the SQL statement inside braces after the #sql token. Listing 11.6 shows two methods using DDL statements:

- The createTable() method executes a CREATE TABLE statement.
- The dropTable() method executes a DROP TABLE statement.

```
01: public void createTable() {
02:   try {
03:     dropTable();
04:     #sql { create table music_cd (
05:              id number(4) primary key,
06:              title varchar2(40) not null,
07:              artist varchar2(40) not null,
08:              create_date date) };
09:   }
10:   catch (Exception e) {
11:     e.printStackTrace();
```

LISTING 11.6 Using DDL statements in SQLJ

```
  12:   }
  13: }
  14:
  15: public void dropTable() {
  16:   int existCount;
  17:   try {
  18:     #sql { select count(*) into :existCount
  19:            from user_tables where table_name = 'MUSIC_CD' };
  20:     if (existCount > 0) {
  21:       #sql { drop table music_cd };
  22:     }
  23:   }
  24:   catch (Exception e) {
  25:     e.printStackTrace();
  26:   }
  27: }
```

LISTING 11.6 *Continued*

Notes on Listing 11.6:

- Line 3 calls the dropTable() method before creating the table.
- Lines 15–27 show the dropTable() method, which uses a SELECT statement to read the Oracle data dictionary table USER_TABLES to determine whether the MUSIC_CD table exists. If the MUSIC_CD table does exist, the existCount is non-zero, and the table is dropped. The example shows that SQLJ statements can be placed inside flow-control statements.

The next example, in Listing 11.7, uses an INSERT statement to add a row to the MUSIC_CD table.

```
01: public void insertCD(int id, String title, String artist) {
02:   try {
03:     #sql { insert into music_cd (id, title, artist, create_date)
04:            values (:id, :title, :artist, sysdate) };
05:     #sql { commit };
06:   }
07:   catch (Exception e) {
08:     e.printStackTrace();
09:   }
10: }
```

LISTING 11.7 Using DML in SQLJ

Notes on Listing 11.7:

- Line 4 of the INSERT statement uses the Java method arguments as input host variables to supply values for the insert statement.
- Line 5 executes a COMMIT statement, assuming that auto-commit has been disabled for the default context used.

11.3.2.1 Storing NULL Values. Java primitive types, such as `byte`, `int`, `short`, `long`, `float`, `double`, and `boolean`, cannot be used to store a database NULL value in a column. To store a NULL value in a column, you must use one of the Java wrapper classes that has been set to a Java `null` reference value, and use the Java object as the host variable in an SQL operation. An example of inserting a NULL value into a numeric column is shown in Listing 11.8.

```
01: public void insertCD(int id, String title, String artist) {
02:   Integer yearReleased = null;
03:   try {
04:     #sql { insert into music_cd
05:              (id, title, artist, year_released, create_date)
06:            values
07:              (:id, :title, :artist, :yearReleased, sysdate) };
08:     #sql { commit };
09:   }
10:   catch (Exception e) {
11:     e.printStackTrace();
12:   }
13: }
```

LISTING 11.8 Inserting a NULL value

Notes on Listing 11.8:

- Line 2 declares a Java object variable yearReleased for the Integer, which is initialized to a `null` value.
- Line 7 in the INSERT statement uses the yearReleased as the input host variable for the year_released column. A NULL value is inserted into the year_released column, because the SQLJ code detects that the object reference is null and converts this into a database NULL value for the column.

The same technique can be used for changing a column value to a NULL, if appropriate, in an UPDATE statement.

To read a column containing a NULL value, you must use a Java object variable defined as an appropriate wrapper class as a host variable. Then, if and only if the Java object reference is not a `null`, you can convert the value in the Java object into its primitive value, At runtime, if you attempt to read a database NULL value into a primitive, the SQLJ statement will fail with the following exception message:

```
sqlj.runtime.SQLNullException: cannot fetch null into primitive data type
```

Reading database NULL values is discussed below in section 11.3.3.

11.3.2.2 Transactions in SQLJ. Transaction control is handled in the same way as storing NULL values, since you perform these tasks in a pure SQL environment. For manual control, you must ensure that the connection context used has auto-commit disabled. To issue a COMMIT, use the SQLJ statement:

```
#sql { COMMIT };
```

To ROLLBACK, execute the following SQLJ statement:

```
#sql { ROLLBACK };
```

You can use the auto-commit feature of the underling JDBC connection if appropriate.

11.3.3 QUERY PROCESSING

In the context of a programming language, data can be queried in two ways. You can retrieve a single row at a time or process a set of rows. Reading a single row requires that you have some way of targeting one and only one row in your query using search criteria, usually an appropriate condition in the WHERE clause of your query.

Processing multiple rows requires creating a cursor structure and stepping through each row of data. In JDBC, you use a ResultSet; in SQLJ, you create an *iterator*.

11.3.3.1 Reading a Single Row. Reading a single row is as simple as embedding a SELECT statement in the code, with the addition of an INTO clause to the query. The syntax for the general structure of a SELECT statement to retrieve a single row is:

```
#sql {  SELECT col(s) INTO variable(s)
        FROM table
        WHERE condition(s) };
```

The number of variables specified in the INTO clause must match the number of columns queried. The data type of each Java variable in the INTO clause must be type-compatible with the SQL type of its corresponding column. The SELECT statement can use any form of Oracle SQL discussed in the earlier chapters of this book. Listing 11.9 shows an example of selecting a single row from the MUSIC_CD table.

```
public void readCd(int id) {
  String title = null;
  String artist = null;
  Integer yearReleased = null;
  java.sql.Date createDate = null;

  try {
    #sql { SELECT title, artist, year_released, create_date
           INTO :title, :artist, :yearReleased, :createDate
           FROM music_cd
           WHERE id = :id };
    System.out.println("Cd: " + id + " " + title + " " +
                       artist + " " + yearReleased + " " +
                       createDate);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

LISTING 11.9 Selecting a single database row

Errors can occur if the search criteria value entered causes no rows to be returned for the query. In this case, the SQLException thrown contains the following message:

```
java.sql.SQLException:
no rows found for select into statement
```

If more than one row is returned by the SELECT statement, the following exception message is generated by the SQLJ Runtime:

```
java.sql.SQLException:
multiple rows found for select into statement
```

If you want to read the Oracle ROWID pseudo-column for each row, include the ROWID in the query and read the value into a Java string variable. The ROWID value can then be used in the WHERE clause of the UPDATE or DELETE statements for fast row access.

11.3.3.2 Reading Multiple Rows Using Iterators. Reading more than one row requires the creation of an SQLJ iterator. An *iterator* is a strongly typed result set. The data types expected for each column in the query are controlled by an SQLJ iterator declaration. There are two types of SQLJ iterators:

- A named iterator declares the names and data type of each column.
- A positional iterator declares only the data type of each column.

The syntax for declaring iterators is:

```
#sql iterator CustomerIterator (int id, String name);

#sql iterator CourierIterator (int, String);
```

The SQLJ Translator creates a Java class for each SQLJ iterator where the class name is derived from the name following the iterator keyword. Each iterator is associated with a query. The number columns selected in the query must match the number of parameters defined inside the parentheses after the iterator name.

To use an iterator, you perform the following steps:

1. Declare the iterator class.
2. Define a Java variable of the iterator class type.
3. Execute a query, compatible with the iterator definition, that returns its results into the Java iterator variable.
4. Use the iterator class methods to fetch each row, and process the column data in each row. The way you process data returned from a named iterator is different from the way you use a positional iterator, but it is conceptually similar to the way you process a JDBC ResultSet.

Here is the general syntax of an iterator and a use example:

```
// 1. Declare the iterator class
#sql iterator MyIterator (int, String);

 :
// 2. Define a variable using the iterator class
MyIterator iter;
```

LISTING 11.10 Syntax and example for using an iterator

```
// 3. Execute a query compatible with the iterator definition
#sql iter = { SELECT id, name FROM table … };

// 4. process the result data using the iter object methods

For example:

#sql iterator CourierIter (int id, String name);

public class Courier {
   public CourierIter getCouriers() {
     CourierIter iter = null;
     try {
       #sql iter = { SELECT id, name FROM courier };
     }
     catch (Exception e) {
       . . .
     }
     return iter;
   }
}
```

LISTING 11.10 *Continued*

Note that the SQLJ Translator generates a Java class for each iterator declared. The methods contained in the generated iterator class enable it to:

- Step through the rows retrieved by a query returning a result set.
- Get column values.

Named and positional iterator declarations cause different method names to be generated in their respective iterator classes.

Declaring and Using a Named Iterator

To declare a named iterator, you must specify a data type and a name for each column value expected from a query that returns a result set to the iterator.

The SQLJ Translator uses the column names specified in the iterator declaration to derive the names of the column value accessor methods in the iterator class. These method names are case-sensitive, as defined by the names in the iterator declaration. The column names in the associated query must match the

names defined in the iterator declaration. However, the column names in the query are treated case-insensitively. The Java data type specified for each iterator column must be type-compatible with its corresponding database column.

Listing 11.10 shows how to create a named iterator to read some of the details from the customer table.

```
#sql iterator CustomerList (int id, String name, String surname);
```

LISTING 11.10 Creating a named iterator for the CUSTOMER table

The SQLJ Translator generates a Java class for the named iterator, called CustomerList, which contains the following methods:

- boolean next()—allows you to fetch the next row of data; returns a true if a row is found, and a false if there are no more rows.
- int id()—returns the customer id as an int, for the current row.
- String name() – returns the customer name as a String.
- String surname() – returns the customer surname as a String.

These methods are used to process the query data. The next() method is common to all named iterators, and the remaining method names depend on the iterator declaration. One method is created for each column name specified in the iterator declaration, which returns a value of the data type declared before the name. The number of columns in the SQL query must be equal to or greater than the number of names listed in the iterator declaration. Additional columns in the query are ignored.

Listing 11.11 shows a method called listCustomers() which uses a modified form of the CustomerList named iterator to display the customer id, name, and email address.

```
01: #sql iterator CustomerList (int id, String fullName,
02:                             String email);
03: public void listCustomers() {
04:   CustomerList custList;
05:
06:   try {
07:     #sql custList = { SELECT name||' '||surname AS fullname,
08:                             email, id
09:                       FROM customer };
10:
11:     while (custList.next()) {
```

LISTING 11.11 Display customer details using a named iterator

```
12:       System.out.println("Customer: " + custList.id() + " " +
13:                          custList.fullName() + " " +
14:                          custList.email());
15:     }
16:     custList.close();
17:   }
18:   catch (Exception e) {
19:     e.printStackTrace();
20:   }
21: }
```

LISTING 11.11 *Continued*

The example in Listing 11.11 highlights some additional points discussed in the notes. Notes for Listing 11.11:

- Lines 1 and 2 declare a named iterator with a comma-separated list of three columns, each preceded by a Java data type.
- Line 7 executes the query that returns the columns for the iterator object. The iterator object variable appears after the #sql token, and before an assignment operator prior to the braces containing the query. The query can also be parameterized with host variables. If the query uses a column expression, then a column alias, with the same name as the corresponding name in the iterator declaration, must be used. In this example, the query uses a concatenated expression of values, the `first_name`, a space, and the `last_name`. The column alias must be specified as `fullname` to match the second iterator column name in line 1.
- Line 11 calls the next() method of the iterator class, as you would with a JDBC ResultSet to process each row of data.
- Lines 12, 13, and 14 print the values returned in each row by calling each of the named iterator methods for the column values defined.

The Java source code generated for the CustomerList class is shown in Listing 11.12.

```
class CustomerList extends sqlj.runtime.ref.ResultSetIterImpl
                   implements sqlj.runtime.NamedIterator
{
  public CustomerList(sqlj.runtime.profile.RTResultSet resultSet)
    throws java.sql.SQLException
  {
```

LISTING 11.12 Generated Java source for a named iterator

```
    super(resultSet);
    idNdx = findColumn("id");
    fullNameNdx = findColumn("fullName");
    emailNdx = findColumn("email");
  }

  public int id() throws java.sql.SQLException {
    return resultSet.getIntNoNull(idNdx);
  }
  private int idNdx;

  public String fullName() throws java.sql.SQLException {
    return resultSet.getString(fullNameNdx);
  }
  private int fullNameNdx;

  public String email() throws java.sql.SQLException {
    return resultSet.getString(emailNdx);
  }
  private int emailNdx;
}
```

LISTING 11.12 *Continued*

The source code in Listing 11.12 is added to the Java source generated for the SQLJ file containing the #sql iterator declaration. The iterator class is subject to Java scoping rules depending on where it is declared. For example, you can declare an iterator as a standalone class or an inner class. The following example illustrates this point:

```
package com.prenhall.OFJP.sqlj;

/* SQLJ iterator generated as normal class with visibility
   defined with "default" access within the package */
#sql iterator CustomerIter (…);

public class OrderEntry {
  // SQLJ iterator declared as an inner class
  #sql iterator CustOrderIter (…);
```

```
  class OrderItem {
    // SQLJ iterator declared as nested inner class
    #sql iterator CourierIter (…);
  }

  public OrderEntry(…) {  // constructor
  }

  public void addItem(…) { }  // instance method
    #sql iterator SaleItemIter (…); // generates a compile time error
}
```

An SQLJ declaration, like these iterators, is invalid inside a method, because the SQLJ Translator does not allow them to be specified in the body of a method. The SQLJ Translator generates an error.

Declaring and Using a Positional Iterator

Declaring a positional iterator is similar to declaring a named iterator, but with a comma-separated list of Java data types without the names. For example:

```
#sql iterator CustomerList (int, String, String);
```

The CustomerList iterator is still considered a strongly typed mechanism. The names of column names / aliases in a query are irrelevant to a positional iterator, as long as the column data type matches the corresponding iterator column declaration.

To process rows with a positional iterator, you first execute an SQLJ FETCH statement, followed by a call to the endFetch() method to test the outcome of the FETCH operation. The SQLJ Translator generates the following methods for a positional iterator:

- boolean endFetch()—returns true if the last SQLJ fetch statement executed returns a row; otherwise, it returns a false.
- getCol<n>() method, where <n> is a number from 1 to the number of column data types specified in the positional iterator definition. Each method returns a value of the data type corresponding to its position.

Listing 11.13 shows the code generated by the SQLJ Translator for a positional iterator:

```
class CustomerIter extends sqlj.runtime.ref.ResultSetIterImpl
                   implements sqlj.runtime.PositionedIterator
{
  public CustomerIter(sqlj.runtime.profile.RTResultSet resultSet)
    throws java.sql.SQLException
  {
    super(resultSet, 3);
  }

  public int getCol1() throws java.sql.SQLException {
    return resultSet.getIntNoNull(1);
  }

  public String getCol2() throws java.sql.SQLException {
    return resultSet.getString(2);
  }

  public String getCol3() throws java.sql.SQLException {
    return resultSet.getString(3);
  }
}
```

LISTING 11.13 Generated Java source for a positional iterator

The endFetch() method is not shown because it is inherited from the sqlj.runtime.ref.ResultSetIterImpl superclass. While you use the getCol<n>() methods to obtain the column values for each row, accessing the row data is accomplished with the SQLJ FETCH statement. An example showing the syntax for a FETCH statement is:

```
package com.prenhall.OFJP.sqlj;
#import sqlj.runtime.*;
#import sqlj.runtime.ref.*;

#sql iterator CustomerIter (int id, String name, String email);

public class RegisterCustomer {

  public void listCustomers() {
    CustomerIter custIter = null;
    int    custId;
    String custName;
    String email;
    #sql custIter = { SELECT id, name, email FROM customer };
```

```
    try {
      while (true) {
        #sql { FETCH :customerIter INTO :custId, :custName, :email };
        if (iter.endFetch()) break;
        :
        // process data here
        :
      }
    }
    catch (Exception e) { . . . }
    finally { . . . }
  }
}
```

The variable name, like `customerIter`, appearing after the FETCH keyword must be of a positional iterator class type. The iterator variable must be preceded by the colon, as is each of the variables after the INTO keyword. The Java host variables after the INTO keyword must be present for each position corresponding with the iterator column definition; and each must be of a compatible data type. Always test whether the fetch operation was successful by executing the iterator endFetch() method.

Listing 11.14 shows an example of the use of a positional iterator for receiving some of the customer details from the customer table.

```
01: #sql iterator CustomerIter (int, String, String);
02:
03: public void getCustomerDetails() {
04:   try {
05:     CustomerIter custList = null;
06:     int id  = 0;
07:     String name = null;
08:     String email = null;
09:
10:     #sql custList = { SELECT id, name ||' '||surname, email
11:                       FROM customer };
12:
13:     do {
14:       #sql { FETCH :custList INTO :id, :name, :email };
15:       if (custList.endFetch()) break;
16:       System.out.println("Cust: " + id +" "+ name +" "+ email);
17:     }
```

LISTING 11.14 Using a positional iterator to read customer details

```
18:     while (true);
19:     custList.close();
20:   }
21:   catch (Exception e) {
22:     e.printStackTrace();
23:   }
24: }
```

LISTING 11.14 *Continued*

Notes for Listing 11.14:

- Line 1 declares the positional iterator class.
- Line 5 defines the iterator variable.
- Lines 6, 7, and 8 define variables for values retrieved from the query.
- Line 10 executes the query returning the result set for the positional iterator object variable custList.
- Lines 13–18 comprise the loop to process each row read in for the specified query.
- Line 14 executes the SQL FETCH statement, receiving one column value per data type position defined in the positional iterator.
- Line 15 tests whether the last FETCH operation was successful, and, if not, the loop is terminated by executing a break statement. Otherwise, the loop continues to process the data.

11.3.3.3 Closing Iterators. Iterators, like the JDBC ResultSet, consume resources, so it is important to close an iterator after processing all the data it returns. Simply call the close() method of the iterator to close it.

11.3.3.4 Reading NULL Values. If any column in the data base table can contain a NULL, then you should read the column value into a Java object reference of a compatible type. This applies specifically to values you wish to receive as a Java primitive type. Primitive types cannot store a Java null value, so you should use the appropriate Java wrapper. For example:

```
#sql iterator DemoIter (int, String);

public void insertNull() {
  try {
    #sql { DROP TABLE demonull };
  }
  catch (Exception e) {}
```

```
  try {
    DemoIter demo = null;

    #sql { CREATE TABLE demonull(id number(4), text varchar2(30)) };
    #sql { INSERT INTO demonull values (1, 'Does not have nulls') };
    #sql { INSERT INTO demonull values (null, 'Has a null') };
    #sql { COMMIT };

    #sql demo = { SELECT * FROM demonull };
    do {
      int idValue = 0;
      String textValue = null;

      #sql { FETCH :demo INTO :idValue, :textValue };
      if (demo.endFetch()) break;
      System.out.println("Row: " + idValue + " " + textValue);
    }
    while (true);
    demo.close();
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

The exception occurs when fetching the second row, and the message generated by SQLJ runtime when attempting to read the NULL valued column is:

```
sqlj.runtime.SQLNullException: cannot fetch null into
primitive data type
```

To avoid this problem:

- Change the iterator definition to read the NULL valued column as a corresponding Java wrapper instead of the primitive type.
- Test the object reference value used to receive the value for a null.
- If the Java object reference is null, then the value in the column was a database NULL value; otherwise, you have an object reference to the value that can be used to convert the value contained in the object into its primitive value by using the appropriate wrapper class method.

The code that shows the suggested changes in bold text is:

```
#sql iterator DemoIter (Integer, String);

public void insertNull() {
  try {
    #sql { DROP TABLE demonull };
  }
  catch (Exception e) {}

  try {
    DemoIter demo = null;

    #sql {
    CREATE TABLE demonull(
       id number(4), text varchar2(30))
    };
    #sql {
    INSERT INTO demonull
    values (1, 'Does not have nulls')
    };
    #sql {
    INSERT INTO demonull
    values (null, 'Has a null')
    };
    #sql { COMMIT };

    #sql demo = { SELECT * FROM demonull };
    do {
      Integer idValue = null;
      String textValue = null;

      #sql { FETCH :demo INTO :idValue, :textValue };
      if (demo.endFetch()) break;
      if (idValue == null) {
        System.out.println("Row: NULL " + textValue);
      }
      else {
        int idVal = idValue.intValue();
        System.out.println(
        "Row: " + idVal + " " + textValue);
      }
    }
    while (true);
    demo.close();
  }
  catch (Exception e) {
```

```
            e.printStackTrace();
          }
        }
```

Two changes were made:

1. The data type of the first column in the iterator definition was changed to an Integer type.
2. The idValue variable was changed to an Integer object.

If the id column has a non-null value, the JDBC driver creates an Integer object for the value; otherwise the object variable, idValue, is assigned a Java null value, that is, no object is created. You can simply test the object reference for `null` to detect whether you have read a NULL database value.

11.3.3.5 Advanced Iterators. In the definition of an iterator, Oracle SQLJ allows a data type for a column to be a ResultSet or another Iterator. This is useful for returning result sets from a nested query, such as when retrieving data in a nested table. In Oracle8i SQL, you can emulate a nested table for related tables by using the CURSOR operator. The following query is an example of an SQL statement that requires an iterator to be defined as a column data type for its associated iterator:

```
SELECT id, name,
       CURSOR (select id, cour_id, total_cost
               FROM cust_order WHERE cust_id = customer.id) orders
FROM customer
WHERE id = :customerId;
```

In the example, the CURSOR operator executes a correlated subquery to return all order records for a specific customer. The subquery returns more than one column value and a set of rows, as if it were a nested table.

The named iterator definition to read the customer id, name, and order rows, using a JDBC ResultSet, is:

```
// Named Iterator
#sql iterator CustOrders (int id, String name, ResultSet orders);
```

The positional iterator to achieve a similar result is:

```
// Positional Iterator
#sql iterator CustOrders (int, String, ResultSet);
```

The query column alias, `orders`, is applied to the CURSOR query so that you can use a named iterator. Without the alias applied to the nested cursor query, you would be forced to use a positional iterator.

Listing 11.15 shows how to process the data in the nested cursor result set.

```
01: #sql iterator CustOrders (int id, String name, ResultSet orders);
02:
03: public void getOrders(int customerId) {
04:   try {
05:     CustOrders custOrders = null;
06:
07:     #sql custOrders = { SELECT id, name,
08:              CURSOR ( SELECT id, cour_id, total_cost
09:                       FROM cust_order
10:                       WHERE cust_id = customer.id) orders
11:            FROM customer WHERE id = :customerId };
12:
13:     while (custOrders.next()) {
14:       int id = custOrders.id();
15:       String name = custOrders.name();
16:       System.out.println("Orders for: " + id + " " + name);
17:       ResultSet ordData = custOrders.orders();
18:       while (ordData.next()) {
19:         System.out.println("\t" +
20:           ordData.getInt(1) + " " +  // order id
21:           ordData.getInt(2) + " " +  // courier id
22:           ordData.getDouble(3));     // total cost
23:       }
24:       ordData.close();
25:     }
26:     custOrders.close();
27:   }
28:   catch (Exception e) {
29:     e.printStackTrace();
30:   }
31: }
```

LISTING 11.15 Using a JDBC ResultSet as an iterator column

Notes for Listing 11.15:

- The example assumes that you have imported the `java.sql` package in addition to the SQLJ packages.
- Line 1 declares a named iterator with the third column data type as a JDBC ResultSet.
- Line 5 creates the object variable for the iterator.
- Lines 7–11 execute the query to return the customer details and the nested result set of orders for the customer.

- Lines 13–25 are the loop to process the rows returned by the iterator, using the iterator methods created by the SQLJ Translator.
- Line 17 gets the ResultSet from the third iterator column, which returns zero or more order rows for the given customer.
- Lines 18–23 loop to read the rows from the nested result set of order data for the given customer.
- When you compile this example with Oracle JDeveloper, it issues a warning that using a ResultSet in an iterator is nonportable, because the ability to use a JDBC result set inside the definition of another iterator is a feature implemented by the Oracle SQLJ Translator.

Instead of using a JDBC ResultSet in the iterator column definition, you can use another iterator. However, to use an iterator nested as a column data type of another iterator, the following applies:

- The nested iterator must be `public` to have the SQLJ Translator create a public iterator class.
- The iterator should be in a separate SQLJ source file because it is declared as public.
- The filename containing the iterator definition must be the same name as the iterator.

For example:

```
// File name: Orders.sqlj

package com.prenhall.OFJP.sqlj;

import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql public iterator Orders (int ordId,
                             Integer courId,
                             Double totalCost);
```

After you have run the SQLJ Translator on the public iterator `Orders`,[7] you can use the `Orders` iterator class name, qualified by its package if required, as the column type in the iterator definition. For example:

[7]If building the SQLJ class using JDeveloper, the SQLJ Translator issues a warning that the public iterator class is public and "should be declared in a file named <Iterator>.java." This cannot be done because the SQLJ Translator works on the SQLJ file. You can disable the warning message by clearing the "Show Warnings" checkbox in the JDeveloper project properties "Compiler" tab.

```
// Declare the application iterator to use a another public iterator

#sql iterator CustOrders (int id, String name,
                          com.prenhall.OFJP.sqlj.Orders orders);
```

Listing 11.16 shows the code that uses the `Orders` iterator nested inside the `CustOrders` iterator.

File: Orders.sqlj

```
package com.prenhall.OFJP.sqlj;

import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql public iterator Orders (int ordId,
                             Integer courId,
                             Double totalCost);
```

File: NestedIterator.sqlj

```
01: package com.prenhall.OFJP.sqlj;
02:
03: import java.sql.*;
04: import sqlj.runtime.*;
05: import sqlj.runtime.ref.*;
06:
07: #sql iterator CustOrders (int id, String name, Orders orders);
08:
09: public class NestedIterator {
10:   :
11:   public void getOrders(int customerId) {
12:     try {
13:       CustOrders custOrders = null;
14:
15:       #sql custOrders = { SELECT id, name,
16:              CURSOR ( SELECT id ordid,
17:                              cour_id courid,
18:                              total_cost totalcost
19:                       FROM cust_order
20:                       WHERE cust_id = customer.id) orders
21:            FROM customer WHERE id = :customerId };
22:
```

LISTING 11.16 Using a nested iterator as an iterator column

```
23:       while (custOrders.next()) {
24:         int id = custOrders.id();
25:         String name = custOrders.name();
26:
27:         System.out.println("Customer: " + id + " " +
28:                            name + " has orders:");
29:         Orders ordData = custOrders.orders();
30:         while (ordData.next()) {
31:           System.out.println("\t" +
32:             ordData.ordId() + " " +
33:             ordData.courId() + " " +
34:             ordData.totalCost());
35:         }
36:         ordData.close();
37:       }
38:       custOrders.close();
39:     }
40:     catch (Exception e) {
41:       e.printStackTrace();
42:     }
43:  }
44: }
```

LISTING 11.16 *Continued*

The code to process the nested iterator data is almost identical in structure to the code in Listing 11.15. The main difference is the methods you use to extract the actual data. The methods you use depend on the type of iterator definition you used; that is, whether it was a named or a positional iterator.

Notes on Listing 11.16:

- Line 7 declares the third iterator column as an `Orders` iterator type.
- Lines 16–20 are the Oracle8i SQL CURSOR subquery providing the data for the nested iterator column. The CURSOR subquery is given an alias of `orders` to match the third iterator column in the `CustOrders` named iterator in line 7.
- Line 21 calls the `orders()` method, from the generated `CustOrders` class, which returns an `Orders` iterator.
- Lines 22–27 print the column values from each row in the `Orders` iterator using the methods generated for that iterator. The `Orders` iterator method names match the alias names for each column in the nested cursor query in lines 16–20.

11.4 PROCESSING ORACLE SQL OBJECT TYPES

In SQLJ, you can also read and write SQL object types using the host variable syntax. The Java type declared for host variables receiving an SQL object must be compatible with the SQL object definition, and is derived from a Java class that can be manually created.[8] However, it is usually more productive to use Oracle JPublisher to generate a custom class from the SQL object definition. This section discusses how to use Oracle JPublisher to generate the classes for Oracle SQL object types, and then shows how to use the generated classes in your SQLJ code to work with Oracle SQL objects.

11.4.1 USING ORACLE JPUBLISHER

Oracle JPublisher is a Java application that reads an Oracle database object and generates the source code for a Java class with the structure and functionality representing it. Oracle JPublisher can generate Java classes for:

- Oracle SQL object types and their methods.
- Oracle SQL Reference types.
- Oracle varying-array and nested-table collections.
- Oracle PL/SQL packages.
- Oracle procedures and functions not defined in a PL/SQL package.

This section explains how to work with JPublisher to generate a Java class for an SQL object type, and how to use the generated Java class either to read an SQL object into your Java application or to save a Java object into its compatible SQL object in an object table or column.

11.4.1.1 Using the Oracle JPublisher Command-Line Utility. Oracle JPublisher is provided as a command-line utility called **jpub** shipped with the Oracle8i software. You can find the utility in the ORACLE_HOME/bin directory. Oracle JPublisher is integrated into Oracle JDeveloper 3.0 or later versions, and can be invoked in a GUI environment. To use Oracle JPublisher you must include in your CLASSPATH the ORACLE_HOME/sqlj/lib/translator.zip file and the JDBC class libraries. The generic syntax of an Oracle JPublisher command line is:

```
jpub –option=value [ -option=value … ]
```

[8]Appropriate Java interfaces must be implemented to manually construct a class to read an SQL object. This technique was covered in the discussion of SQLData and CustomDatum interfaces in Chapter 10.

The `jpub` command is followed by one or more `option=value` pairs. Spaces are not allowed between the minus sign option name, the equals sign, and the value. The options control the rules that govern the generation of the Java class. To preserve Java naming conventions, Oracle JPublisher options let you specify the Java class name generated and its corresponding SQL object type. This is achieved by using JPublisher command-line options or via options specified in a properties file. For example:

```
jpub -user=bookstore/bookstore
     –url=jdbc:oracle:thin:@localhost:1521:ORA815
     –sql=customer_t:Customer
     -package=com.prenhall.OFJP.jpub
```

This example generates a `Customer.java` class file for the `customer_t` SQL object in a package called `com.prenhall.OFJP.jpub`.

If you use an input file, you can also control the names of the methods generated in the Java class for each attribute found in the SQL object definition. For example:

With a customer_t Object type defined as:

```
CREATE TYPE customer_t AS OBJECT (
  id        NUMBER(6),
  name      VARCHAR2(30),
  surname   VARCHAR2(40)
);
```

Using a properties file called of myprops.jpub containing:

```
jpub.user=bookstore/bookstore
jpub.url=jdbc:oracle:thin:@localhost:1521:ORA815
jpub.sql=customer_t
jpub.package=com.prenhall.OFJP.jpub
jpub.input=translate.txt
```

And the file translate.txt containing:

```
SQL customer_t AS Customer
   TRANSLATE name AS FirstName,
             surname AS Surname
```

The JPublisher command is:

```
Jpub –props=myprops.jpub
```

The preceding example creates a `Customer.java` file with an accessor method for each attribute. The default accessor method name for the `id` attribute is called getId(). However, the accessor method generated for the `name` attribute is called getFirstName(), as controlled by the input-file commands, as is the method getSurname() for the `surname` attribute. The input option provides a file of command-to-control method name generation.

11.4.1.2 JPublisher Command-Line Options. Table 11.2 lists some of the common JPublisher command-line options and their values. It shows default values (if applicable) in bold text, and required values in italic text.

The same options can be specified more than once, with the last occurrence overriding any previous settings on the command line or in the properties file. Options in the property file are processed as if they were entered on the command line where the `props` option is used.

TABLE 11.2 Oracle JPublisher command-line options

OPTION	VALUES	DESCRIPTION
user	*<username>/password>*	Required option to select the user name and password for the database user who owns the SQL object type definitions.
url	*jdbc-url*	
	jdbc:oracle:oci8:@	The URL can be for other JDBC drivers supported by the vendor. You can use the Oracle thin driver with JPublisher if you do not have SQL*net/Net8 client software installed on your development platform.
package	*Package-name*	The Java package name for the generated Java code. This creates a subdirectory structure based on the package name in the directory specified by the "dir" option.
sql	*sql-type-name:java-class-name*	Sets the name Java class for the corresponding SQL object type.
props	*Filename*	Specifies the name of the properties file containing additional JPublisher options.
methods	all, named, none	
	Using an input file with the "named" value allows you to specify which SQL methods are mapped, and all others are ignored. A value of true is a synonym for all, and false is a synonym for none.	Specifies whether the generated Java class contains wrapper methods for those found in the SQL object type or PL/SQL package.
input	*Filename*	Specifies a file name that contains commands controlling how SQL object types, PL/SQL packages, and subcomponents are translated.

11.4.1.3 Files Generated by JPublisher. The files generated by Oracle JPublisher depend on how the –sql option is used. For example, if you specify:

```
jpub –sql=oracle-type:ClassName . . .
```

then JPublisher generates the following files:

- An SQLJ or Java file called *`ClassName`*`.sqlj` or *`ClassName`*`.java`. The SQLJ file is generated if you specify the `–methods=true` command-line option, otherwise the `.java` file is generated.
- A Java file called for a *`ClassName`*`REF.java` to work with a database REF to an SQL object type. This file is only generated when the *oracle-type* specifies an SQL object type name.
- If the *oracle-type* is a PL/SQL package name or the keyword `TOPLEVEL`, you must also include the `–methods=true` option.

The Java data types generated for attributes, method return values, and method arguments are influenced by the `-mapping` options, or via the four options: `-builtintypes`, `-usertypes`, `-lobtypes`, and `-numbertypes`. Mapping options are mentioned to highlight that you can control the data types generated for Java variables and methods, but it would be too much of a digression to discuss them in any detail.

11.4.1.4 Using a Properties File. If you use a properties file specified in the `–props=`*`filename`* option, each line of the property file specifies a property whose name is prefixed with **`jpub.`** The option is followed by an equals sign and the value. For example:

```
jpub.user=bookstore/bookstore
jpub.sql=customer_t
jpub.mapping=jdbc
jpub.package=com.prenhall.OFJP.jpub
```

The equivalent command line is:

```
jpub -user=bookstore/bookstore –sql=customer_t -mapping=jdbc
     -package=com.prenhall.OFJP.jpub
```

Options not prefixed with `jpub` are ignored.

> **Note**
>
> The command line is continued according to the rules of the operating system or command-line handler.

11.4.1.5 Controlling the Generation of Class Names. To control the names generated in your Java class and for each accessor method created for the SQL object type attributes, use the `-input` option, which specifies a file name containing one or more *translation statements* that control the name generation.

A translation statement begins with the keyword SQL followed by the name of the database structure to be translated and additional instructions introduced with the keywords GENERATE, AS, and TRANSLATE.[9] The abbreviated syntax of a translation statement is:

```
SQL name [AS java-name-2]
  [TRANSLATE database-member-name AS simple-java-name
          [, database-member-name AS simple-java-name … ]
```

The ***name*** entered after the SQL keyword can be specified as:

- An SQL object type name or a PL/SQL package name.
- An SQL object type or a PL/SQL package prefixed with a specific database schema name.
- The keyword TOPLEVEL, which specifies that all PL/SQL procedures and functions in the current schema are to be translated as methods into the same Java class. The keyword TOPLEVEL is a reserved word, and can be prefixed with a database schema name.

For example, if the input file contains:

```
SQL customer_t AS Customer
```

`Customer` is used as the file and class name generated for the SQL object type called `customer_t`. The name after the AS keyword is case-sensitive. For example:

```
SQL customer_t AS CustoMER
```

This would generate a file called `CustoMER.java` and the class name would appear as follows:

```
public class CustoMER {
    :
  }
```

[9]The SQL keyword is the preferred command, but can be replaced with the keyword TYPE. However, the TYPE keyword may be deprecated in future versions of Oracle JPublisher.

Therefore, you must take care with the case of characters entered for the Java class name.

11.4.1.6 Controlling the Generation of Method Names. The TRANSLATE command in the input file specifies how to convert attributes in the SQL object type into Java accessor method names. Oracle JPublisher creates a get and set method for each attribute found in the SQL object type definition. The "get" and "set" keywords, in lowercase, are prefixed to a Java method name specified TRANSLATE command. For example:

```
SQL customer_ot AS Customer
  TRANSLATE name AS FirstName,
             surname AS Surname
```

The Java class file generated is called Customer.java, and contains methods with the signatures shown in the following code snippet:

```
public class Customer {
  :

  public void setFirstName(String FirstName) throws SQLException {
    :
  }

  public String getFirstName() throws SQLException {
    :
  }

  public void setSurname(String Surname) throws SQLException {
    :
  }

  public String getSurname() throws SQLException {
    :
  }
}
```

Note that the argument names preserve the case of the java attribute names specified in the TRANSLATE command option.

11.4.1.7 Using the Command Line to Control Class Name Generation. The `-sql` option is a shortcut alternative to the `-input` option for controlling the generation of a class name. The `-sql` option can be specified as:

```
-sql=type-name
```

This creates a Java class with the same name as the type name. However, underscore characters in the type-name are excluded from the resulting Java class name, and each word is capitalized. For example:

```
jpub –sql=customer_t
```

This creates two Java class files named CustomerT.java, and CustomerTRef.java.

Alternatively, you can use the `–sql` option to name the Java class as follows:

```
-sql=type-name:Javaclass
```

This creates a Java class of the name you specify. The Java class names if entered are case-sensitive, but the database type-name is not case-sensitive. More than one `type-name:java-class` combination can be entered, separated by commas and without spaces. For example:

```
jpub –sql=custorder_t:CustomerOrder,courier_t:Courier
```

This JPublisher command will generate four Java source files:

- CustomerOrder.java and CustomerOrderRef.java for the `customer_t` SQL object type.
- Courier.java file and CourierRef.java for the `courier_t` SQL object type.

11.4.2 USING THE CLASSES GENERATED BY JPUBLISHER

The SQLJ or Java class files generated by JPublisher can be used in your SQLJ or JDBC code. The examples in this chapter focus on using the generated classes in SQLJ. If you want to use them in JDBC, follow the examples in Chapter 10 2, which discusses using the SQLData and CustomDatum interfaces for object types. The examples in SQLJ are based on an SQL object type called `customer_t`. The `customer_t` object type definition is:

```
01: CREATE TYPE customer_t AS OBJECT (
02:   ID                          NUMBER(6),
03:   NAME                      VARCHAR2(30),
04:   SURNAME                   VARCHAR2(30),
05:   EMAIL                     VARCHAR2(50),
06:   PASSWORD                  VARCHAR2(10),
07:   CREDIT_CARD_TYPE          VARCHAR2(10),
```

```
08:   CREDIT_CARD_NUMBER      VARCHAR2(20),
09:   MONTH_EXPIRED           VARCHAR2(2),
10:   YEAR_EXPIRED            VARCHAR2(2)
11: );
12:
13: -- Create Object Table
14: create table customer of customer_t;
15:
16: // Create Relational Table with Object Column
17: create table best_cust (
18:   ID    NUMBER(4) CONSTRAINT best_cust_pk PRIMARY KEY,
19:   CUST  CUSTOMER_T
20: );
21:
22: // Create Relational Table with REF to Object
23: create table reg_cust (
24:   ID          NUMBER(4) CONSTRAINT reg_cust_pk PRIMARY KEY,
25:   CUSTREF     REF CUSTOMER_T
26: );
```

- Line 14 creates an object table of customers
- Lines 17–20 create the BEST_CUST table, which is used for reading or updating an object column.
- Lines 23–26 create the REG_CUST, which is used for inserting and reading a SQL reference to an object column.

The Oracle JPublisher command line used to generate the Customer.java and CustomerRef.java file is:

```
01: jpub -sql=customer_t:Customer
02:       -url=jdbc:oracle:thin:@localhost:1521:ORA815
03:       -user=obook/obook
```

The `jpub` command and options have been shown on three lines for clarity, but it should all be entered on one line. This JPublisher command creates two files:

1. Customer.java (see Listing 11.17)[10]
2. CustomerRef.java (see Listing 11.18)

[10]The file is called Customer.sqlj if you invoke JPublisher from JDeveloper.

The generated Customer.java file uses the CustomDatum and CustomDatumFactory interfaces, making the code not portable to other database environments.

```
01: public class Customer
02:   implements CustomDatum, CustomDatumFactory
03: {
04:   public static final String _SQL_NAME = "OBOOK.CUSTOMER_T";
05:   public static final int _SQL_TYPECODE = OracleTypes.STRUCT;
06:
07:   /* constructors */
08:   public Customer() { ... }
09:   public Customer(ConnectionContext c)
10:     throws SQLException { ... }
11:   public Customer(Connection c)
12:     throws SQLException { ... }
13:
14:   /* CustomDatum interface */
15:   public Datum toDatum(OracleConnection c)
16:      throws SQLException { ... }
17:
18:   /* CustomDatumFactory interface */
19:   public CustomDatum create(Datum d, int sqlType)
20:     throws SQLException { ... }
21:
22:   /* shallow copy method: give object same attributes as args */
23:   void shallowCopy(Customer d) throws SQLException {
24:     _struct = d._struct;
25:   }
26:
27:   /* accessor methods */
28:   public BigDecimal getId() throws SQLException
29:   { return (BigDecimal) _struct.getAttribute(0); }
30:
31:   public void setId(BigDecimal id) throws SQLException
32:   { _struct.setAttribute(0, id); }
33:
34:   public String getName() throws SQLException
35:   { return (String) _struct.getAttribute(1); }
36:
37:   public void setName(String name) throws SQLException
38:   { _struct.setAttribute(1, name); }
39:
40:   // other accessor methods ...
41: }
```

LISTING 11.17 JPublisher-generated Customer.java class

For brevity, most of the generated code for the Customer.java source has been omitted, such as imports and method bodies. Some of the method signatures have been kept, to highlight the class structure.

Notes on Listing 11.17:

- Line 5 identifies the Java object as being an OracleTypes.STRUCT for the JDBC layer to manage the type mapping.
- Lines 8–12 are the constructors for the class; the no-argument constructor must be present.
- Lines 14–16 show the implementation of the CustomDatum.toDatum() method.
- Lines 18–20 implement the CustomDatumFactory.create() method to instantiate a Customer object.
- Lines 27–41 are the class getter and setter methods, which manage the state of each Customer instance.

The Customer.java class should be compiled first, because it is referenced in the CustomerRef.java class.

```
01: import java.sql.SQLException;
02: import oracle.jdbc.driver.OracleConnection;
03: import oracle.jdbc.driver.OracleTypes;
04: import oracle.sql.CustomDatum;
05: import oracle.sql.CustomDatumFactory;
06: import oracle.sql.Datum;
07: import oracle.sql.REF;
08: import oracle.sql.STRUCT;
09:
10: public class CustomerRef
11:    implements CustomDatum, CustomDatumFactory
12: {
13:   public static final String _SQL_BASETYPE = "OBOOK.CUSTOMER_T";
14:   public static final int _SQL_TYPECODE = OracleTypes.REF;
15:
16:   REF _ref;
17:
18:   static final
19:      CustomerRef _CustomerRefFactory = new CustomerRef();
20:
21:   public static CustomDatumFactory getFactory() {
22:     return _CustomerRefFactory;
23:   }
24:
```

LISTING 11.18 JPublisher-generated CustomerRef.java class

```
25:   public CustomerRef() { // constructor
26:   }
27:
28:   /* CustomDatum interface */
29:   public Datum toDatum(OracleConnection c)
30:                               throws SQLException {
31:     return _ref;
32:   }
33:
34:   /* CustomDatumFactory interface */
35:   public CustomDatum create(Datum d, int sqlType)
36:                                    throws SQLException {
37:     if (d == null) return null;
38:     CustomerRef r = new CustomerRef();
39:     r._ref = (REF) d;
40:     return r;
41:   }
42:
43:   public Customer getValue() throws SQLException {
44:      return (Customer) Customer.getFactory().create(
45:        (Datum)  _ref.getValue(), OracleTypes.REF);
46:   }
47:
48:   public void setValue(Customer c) throws SQLException {
49:     _ref.setValue((STRUCT) c.toDatum(_ref.getConnection()));
50:   }
51: }
```

LISTING 11.18 *Continued*

The CustomerRef class provides Java developers with a way to work with SQL object type REF values.

Notes on Listing 11.18:

- Line 14 identifies the object instance as an OracleTypes.REF for the JDBC layer type mapping.
- Line 29 implements the toDatum() method for the CustomDatum, and line 35 implements the create() method for the CustomDatumFactory interface.
- Lines 43–46 show the getValue() method, which you can use to obtain an instance of the Customer via the CustomerRef object.
- Lines 48–50 show the setValue() method, which provides you with a means to write a reference to a Customer object previously obtained from the database.

Listings 11.17 and 11.18 provide you with a quick look under the hood at the classes generated by JPublisher. The JPublisher tools save a great deal of coding effort, and eliminate an error-prone manual process, to create custom classes for Oracle SQL object types and an object type REF.

11.4.2.1 Selecting Oracle SQL Objects in SQLJ. To select an SQL object into your Java application, you create an SQLJ SELECT statement to retrieve the data from an object table or column, and store the SQL object value in a Java variable declared with the class name generated by JPublisher for the corresponding SQL object type.

Listing 11.19 shows an example that queries all customer instances from an object table, using a named SQLJ iterator.

```
01: package com.prenhall.OFJP.sqlj;
02:
03: import sqlj.runtime.*;
04: import sqlj.runtime.ref.*;
05: import java.math.BigDecimal;
06:
07: public class ManageCustomer {
08:
09:   public ManageCustomer(String url) { ... }
10:
11:   #sql iterator List(Customer cust);
12:
13:   public void listCustomers() {
14:      List     customers;
15:
16:      try {
17:        #sql customers = { select value(c) cust from customer c };
18:        while (customers.next()) {
19:          Customer theCust = customers.cust();
20:          System.out.println(theCust.getId() + " " +
21:                             theCust.getName() + " " +
22:                             theCust.getSurname());
23:        }
24:        customers.close();
25:      }
26:      catch (Exception e) {
27:        e.printStackTrace();
28:      }
29:   }
30: }
```

LISTING 11.19 Selecting an SQL object from an object table

Notes for listing 11.19:

- Line 9 is the skeleton for the ManageCustomer constructor. The default connection context is initialized in the constructor.
- Line 11 defines a named iterator called `List`, which defines the column data type as `Customer` and name as `cust`.
- Line 14 declares a local iterator variable called `customers`.
- Line 17 executes the SELECT statement to read the customer object instances from the table, and returns the result set to the `customers` iterator.
- Line 19 obtains the Customer object instance from the iterator cust() method.
- Lines 20–22 call some of the getter methods generated by JPublisher to display some of the customer details.[11]

Listing 11.20 shows how to use a positional iterator to fetch a specific customer instance from the object table.

```
01: package com.prenhall.OFJP.sqlj;
02:
03: import sqlj.runtime.*;
04: import sqlj.runtime.ref.*;
05: import java.math.BigDecimal;
06:
07: public class ManageCustomer {
08:
09:   public ManageCustomer(String url) { ... }
10:
11:   #sql iterator PList(Customer);
12:
13:   public void getCustomer(int id) {
14:     PList iter;
15:     Customer c = null;
16:
17:     try {
18:        #sql iter = { select value(c)
19:                      from customer c
```

LISTING 11.20 Reading an Object Type using a positional iterator

[11] In addition to accessor methods, JPublisher can also generate Java wrapper methods for each member method in the SQL object type definition if you use the JPublisher `–methods` command-line option.

```
20:                       where c.id = :id };
21:        #sql { fetch :iter into :c };
22:        if (!iter.endFetch()) {
23:          System.out.println(c.getId() + " " +
24:                        c.getName() + " " +
25:                        c.getSurname());
26:        }
27:        else {
28:          System.out.println(
29:             "Customer with " + id + " does not exist");
30:        }
31:     }
32:     catch (Exception e) {
33:       e.printStackTrace();
34:     }
35:   }
36: }
```

LISTING 11.20 *Continued*

Notes on Listing 11.20:

- Line 11 declares the positional iterator class called PList.
- Line 18 assigns the result set from the select statement to the iterator variable, declared in line 14.
- Line 21 fetches an SQL CUSTOMER_T object into the Java object reference for a Customer.
- Lines 23–25 call the getter methods from the Customer class to display some of the attribute values.

The SQLJ code for retrieving the SQL object is quite simple, because the complexity of converting an Oracle SQL object type into a Java object is managed by the code generated by Oracle JPublisher.

11.4.2.2 Inserting, Updating, and Deleting an Oracle SQL Object. Inserting or updating an SQL object with a new Java object instance data is a three-step process:

1. Instantiate the Java object from the class generated by JPublisher.
2. Call the various set methods in the object to set the attributes.
3. Bind the object reference variable in an SQLJ insert or update statement.

Listing 11.21 is an example of creating a customer object, calling the setter methods to define the object state, and then inserting the data into an SQL object instance in an object table. The example also shows how to insert an object into an object column in a relational table.

```
01: package com.prenhall.OFJP.sqlj;
02:
03: import sqlj.runtime.*;
04: import sqlj.runtime.ref.*;
05: import java.math.BigDecimal;
06:
07: public class ManageCustomer {
08:
09:   public ManageCustomer(String url) { ... }
10:
11:   public  void addCustomer(int id, String name,
12:                            String surname, String cardNumber) {
13:      try {
14:        Customer c = new Customer();
15:        c.setId(new BigDecimal(id));
16:        c.setName(name);
17:        c.setSurname(surname);
18:        String email = name.substring(1,2).toUpperCase() +
19:                       "." + surname + "@ozemail.com.au";
20:        c.setEmail(email);
21:        c.setPassword("welcome");
22:        c.setCreditCardType("AMEX");
23:        c.setCreditCardNumber(cardNumber);
24:        c.setMonthExpired("02");
25:        c.setYearExpired("02");
26:
27:        #sql { insert into customer values (:c) };
28:        #sql { insert into best_cust values (:id, :c) };
29:
30:        #sql { commit };
31:      }
32:      catch (Exception e) {
33:        e.printStackTrace();
34:      }
35:   }
36: }
```

LISTING 11.21 Inserting Java objects into an object table or column

This example is somewhat contrived and explicit to show the creation of a Customer object and the operations necessary to INSERT it into an object table and an object column. The code could be written to receive a reference to Customer object, which would be created by the caller of the addCustomer() method. The addCustomer() method signature would then be:

```
public void addCustomer(Customer newCustomer) { . . . }
```

Notes on Listing 11.21:

- Line 14 instantiates the Customer object using its no-argument constructor.
- Lines 15–25 call the set accessor methods of the Customer class to set the state of the customer object.
- Line 27 inserts the object into the customer object table.
- Line 28 inserts the object into an object column of a relational table.

The INSERT statement adding the object into the object table could be written in two other ways. With one, you insert the data using standard SQL INSERT syntax:

```
#sql { insert into customer (id, name, surname, email,
          password, credit_card_type, credit_card_number,
          month_expired, year_expired)
       values (
           :(c.getId()),
           :(c.getName()),
           :(c.getSurname()),
           :(c.getEmail()),
           :(c.getPassword()),
           :(c.getCreditCardType()),
           :(c.getCreditCardNumber()),
           :(c.getMonthExpired()),
           :(c.getYearExpired())
       )
     };
```

Alternatively, you can insert the data using the SQL object type constructor:

```
          #sql { insert into customer values (
              CUSTOMER_T(:(c.getId()),
                         :(c.getName()),
                         :(c.getSurname()),
                         :(c.getEmail()),
```

```
                        :(c.getPassword()),
                        :(c.getCreditCardType()),
                        :(c.getCreditCardNumber()),
                        :(c.getMonthExpired()),
                        :(c.getYearExpired())
                        ))
        };
```

The preceding examples show that, instead of obtaining the SQL statement values from a Java variable, you can place a colon before a host expression enclosed between brackets. In these examples, each host expression calls a Java method to return a result. The result is supplied as the value for the target columns in the SQL statement. In the example, all SQL attributes values are obtained directly from the Customer object in the Java code.

Listing 11.22 provides an example of executing an SQL UPDATE statement on an object column in the SQLJ application.

```
01: package com.prenhall.OFJP.sqlj;
02:
03: import sqlj.runtime.*;
04: import sqlj.runtime.ref.*;
05: import java.math.BigDecimal;
06:
07: public class ManageCustomer {
08:
09:   public ManageCustomer(String url) { ... }
10:
11:   public void changeCustomer(int id) {
12:      try {
13:        Customer aCust = new Customer();
14:        aCust.setId(new BigDecimal(98));
15:        aCust.setName("Xak");
16:        aCust.setSurname("Idran");
17:        aCust.setEmail("Z.Idran@amil.com.za");
18:        aCust.setPassword("welkom");
19:        aCust.setCreditCardType("VISA");
20:        aCust.setCreditCardNumber("2230414134093333");
21:        aCust.setMonthExpired("12");
22:        aCust.setYearExpired("01");
23:
24:        #sql { update best_cust
```

LISTING 11.22 Updating an SQL object in an object column

```
25:               set cust = :aCust where id = :id  };
26:
27:         #sql { commit };
28:       }
29:       catch (Exception e) {
30:         e.printStackTrace();
31:       }
32:   }
33: }
```

LISTING 11.22 *Continued*

Notes on Listing 11.22:

- Lines 13–22 instantiate the object and set the state of each instance variable by calling the Customer setter methods.
- Lines 24 and 25 show the update statement used to change the object instance in the `cust` column of the `best_cust` table for a specified customer id. The original instance in the `cust` column is overwritten by the new instance.

You cannot use an UPDATE statement to replace an entire object instance in an object table. However, you can modify the attribute values of an existing object instance by using a standard SQL UPDATE statement.

You can delete an object instance by executing any DELETE statement with a condition to target the specific instance. To remove an object instance from an object column, you set the column to a NULL using an UPDATE statement, provided the object column allows a NULL value. For example:

```
package com.prenhall.OFJP.sqlj;

import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.math.BigDecimal;

public class ManageCustomer {

public ManageCustomer(String url) { ... }

  public void removeCustomer(int id) {
    try {
      #sql { update best_cust
             set cust = NULL where id = :id  };
```

```
        #sql { commit };
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
```

11.4.2.3 Extending Classes Generated by JPublisher. The best way to modify the classes generated by JPublisher is to create a subclass of the generated class, and add additional constructors, attributes, and methods to it. The less attractive alternative is to directly modify the generated class. This risks loss of code if you need to regenerate the Java class, due to changes made to the associated SQL object type.

If you create a subclass from the Java class generated by Oracle JPublisher, and subsequently modify the base object type, you must use the Oracle JPublisher input file with the GENERATE AS syntax for the translator command. For example, if the properties file myprops.jpub contains:

```
jpub.user=bookstore/bookstore
jpub.url=jdbc:oracle:thin:@localhost:1521:ORA815
```

where the translate.txt file contains:

```
SQL customer_t
   GENERATE CustomerImpl
    AS MyCustomer
```

you can use the following Oracle JPublisher command line:

```
jpub –props=myprops.jpub –input=translate.txt
```

This example creates two Java files, CustomerImpl.java and CustomerImplRef.java. However, Oracle JPublisher does not generate the file called MyCustomer.java. MyCustomer is added to the translator command after the keyword AS, when using the GENERATE keyword, to prevent JPublisher from creating MyCustomer.java file. Use this technique if you have manually created MyCustomer.java as a subclass of CustomerImpl.java and want to preserve the extension you have made. You can safely regenerate the CustomerImpl class and inherit the appropriate changes.

The shortcut Oracle JPublisher command to perform the same task, using the properties file and not the input translator file, is:

```
jpub -props=p1.jpub -sql=customer_t:CustomerImpl:MyCustomer
```

11.4.2.4 Compile and Runtime SQL Checks. It is usually a good idea to invoke translation-time SQL syntax and validity checks. The compile-time SQL checking is achieved with the SQLJ command line, by providing:

- A username and password with the `–user` option.
- A database connection string using the `-url` option.

However, testing done with the SQLJ Translator shows that if a value assigned to an instance variable in the Java object is too large for the precision of the associated attribute defined in the SQL object type, then a runtime exception would be thrown.[12] Since the SQLJ translation-syntax checking cannot detect the attribute size and bound problems that may occur at runtime, particularly with String data types, you should add the necessary data-validation code before you execute the SQL statement.

11.4.2.5 Using the Java Class for an Object Type REF. As previously stated, for each SQL object type generated by the Oracle JPublisher utility, you get a Java class representing the object REF for the object type. Listing 11.23 demonstrates how you can query and use an object REF type.

```
01: package com.prenhall.OFJP.sqlj;
02:
03: import sqlj.runtime.*;
04: import sqlj.runtime.ref.*;
05: import java.math.BigDecimal;
06:
07: public class ManageCustomer {
08:
09:   public ManageCustomer(String url) { ... }
10:
11:   public void getCustFromRef(int id) {
12:      try {
13:        CustomerRef aCustRef = null;
14:        Customer c = null;
15:
```

LISTING 11.23 Querying an object type REF column

[12]The tests were done with the SQLJ Translator shipped with JDeveloper 3.0.

```
16:        #sql { select ref(c) into :aCustRef
17:               from customer c
18:               where "ID" = :id };
19:
20:        c = aCustRef.getValue();
21:        System.out.println(c.getId() + " " +
22:                           c.getName() + " " +
23:                           c.getSurname());
24:
25:        #sql { insert into reg_cust (id, cust_ref)
26:               values (:id, :aCustRef) };
27:
28:        #sql { commit };
29:      }
30:      catch (Exception e) {
31:        e.printStackTrace();
32:      }
33:   }
34: }
```

LISTING 11.23 *Continued*

Notes on Listing 11.23:

- Line 13 declares the Java variable for the REF of a Customer object.
- Line 14 creates the variable for the Customer object referenced by the CustomerRef object.
- Lines 16–18 execute the Oracle select statement to read an object type REF value into the CustomerRef object. The `id` column is quoted in uppercase to work around a problem with using JDeveloper 3.0 code editor.
- Line 20 calls the getValue() method of the Object REF component to acquire the Customer object instance referenced by the REF value.
- Lines 25 and 26 show an insert statement for storing the CustomerRef into a column `cust_ref` defined as a REF to the `customer_t` in the `reg_cust` table.

The safest way to insert an Object REF value into a REF column in the database is to:

- Select the Object REF value into the Java CustomerRef class from a valid CUSTOMER_T object instance.
- Execute an INSERT or UPDATE statement with the CustomerRef object retrieved, as shown in Listing 11.23.

You can see from the simplicity of the examples used that it is very easy to work with Oracle object types in Java code. There is a natural one-to-one match between the Java object and the SQL object, and using the JPublisher tool simplifies the process of creating the Java class for an SQL object.

11.5 PROCESSING SQL COLLECTIONS

Oracle varying-array and nested-table object types are collections that can be read into, and written from, your Java application. The Java classes compatible with the collection data type definition must first be generated using JPublisher. The following example uses:

- A PERSON_T object type, which contains PL/SQL methods.
- A varying-array collection, called PERSON_VA_T of PERSON_T objects.
- A nested-table collection, called PERSON_NT_T of PERSON_T objects.

The steps in generating and using these SQL objects in Java are shown in the following order:

1. Creating the PERSON_T Object type, Collections, and tables.
2. Generating the Java classes for PERSON_T object type and collections.
3. Using the object type and collections.

11.5.1 CREATING THE SQL COLLECTIONS AND TABLES

Listing 11.24a shows the SQL code used to create the PERSON_T object type. Listing 11.24b shows the creation of the PERSON_VA_T and PERSON_NT_T collections, as well as the following database tables:

- A TEAM table for a nested table of team members.
- A FAMILY table containing a varying array of members.

```
CREATE OR REPLACE TYPE person_t AS OBJECT (
  id          NUMBER(6),
  firstName   VARCHAR2(20),
  lastName    VARCHAR2(20),
  birthDate   date,
  MEMBER FUNCTION getAge RETURN NUMBER,
  MEMBER PROCEDURE setFirstName(newName VARCHAR2),
```

Listing 11.24a Creating PERSON_T SQL object and methods

```
  MEMBER PROCEDURE setLastName(newName VARCHAR2),
  MEMBER PROCEDURE setBirthDate(newDate DATE),
  MEMBER FUNCTION getName RETURN VARCHAR2,
  MEMBER FUNCTION toString RETURN VARCHAR2
);
/

CREATE OR REPLACE TYPE BODY person_t IS
  member function getAge return number is
  begin
    return round(months_between(trunc(sysdate),self.birthDate)/12,2);
  end;

  member procedure setFirstName(newName varchar2) is
  begin
     self.firstName := initcap(newName);
  end;

  member procedure setLastName(newName varchar2) is
  begin
     self.lastName := initcap(newName);
  end;

  member procedure setBirthDate(newDate date) is
  begin
    self.birthDate := trunc(newDate);
  end;

  member function getName return varchar2 is
  begin
    return firstName||' '||lastName;
  end;

  member function toString return varchar2 is
  begin
     return id ||': '||self.getName||' ('||self.getAge||')';
  end;

END;
/
```

Listing 11.24a *Continued*

Listing 11.24a shows a simple object type structure that contains several methods to operate on the object instance in application memory, whether a PL/SQL application or a Java application. The methods defined for the PERSON_T object are:

- Function `getAge` returns the current age of the person. The age is determined by calculating the months between the person's birth date and the current date and time,[13] and dividing the value by twelve using Oracle data arithmetic.
- Procedure `setFirstName` changes the first-name attribute of the SQL object.
- Procedure `setLastName` changes the last-name attribute of a person object.
- Procedure `setBirthDate` changes the birth date in the object.
- Function `getName` returns the concatenated result of the person's first and last names.
- Function `toString` returns a concatenated string representation of the person object attributes.

The method names have been entered using Java naming conventions, but the Oracle database treats each name in a case-insensitive way. Note that the names are stored in uppercase form in the Oracle data dictionary. The method names chosen minimized the need to specify an input file with the Oracle JPublisher command line that was used to create the corresponding Java class. The method names defined in the type body are called through wrapper methods in the Java class generated by JPublisher by specifying the `–methods` command-line option.

The SQL statements used to create the varying-array, nested-table, and database tables based on these types are shown in Listing 11.24b.

```
-- Create the nested table type definition
CREATE TYPE person_nt_t AS TABLE OF person_t;
/

-- Create the varying array type definition
CREATE TYPE person_va_t AS VARRAY(5) OF person_t;
/

-- Create the family table using the varying array for members
```

LISTING 11.24b Creating SQL collections and tables for PERSON_T

[13]The Oracle SQL date functions do not provide a function to determine the years between two dates.

```
CREATE TABLE FAMILY (
  ID        NUMBER(4) CONSTRAINT family_pk PRIMARY KEY,
  MEMBERS   PERSON_VA_T
);

-- Create the team table using the nested table for members
CREATE TABLE TEAM (
  id        NUMBER(4) CONSTRAINT team_pk PRIMARY KEY,
  members   PERSON_NT_T)
NESTED TABLE members STORE AS TEAM_MEMBERS;
```

LISTING 11.24b *Continued*

Having created the SQL types and tables using Oracle JPublisher either as a standalone command-line tool or invoked through JDeveloper, you create Java classes for the object types you wish to work with in your Java applications.

11.5.2 GENERATING JAVA CLASSES FOR SQL COLLECTIONS

Listing 11.25 shows the JPublisher command line and input file used to generate the Java classes for the PERSON_T, PERSON_VA_T object types. A Java class for PERSON_NT_T also needs to be created for manipulation of the data in a nested table. Section 11.5.3, "Accessing the SQL Collections from Java," gives an example of using the generated collection classes.

```
01: jpub -sql=person_t:Person,person_va_t:PersonArray
02:      -url=jdbc:oracle:thin:@localhost:1521:ORA815
03:      -user=bookstore/bookstore
```

LISTING 11.25 Generating the Collection Java class with JPublisher

Notes on Listing 11.25:

The entire command line is entered on one line, but is split into three lines for presentation here.

- Line 1 converts the PERSON_T into a Person.java, and PERSON_VA_T into PersonArray, which creates methods to manage a collection of Person objects.
- Lines 2 and 3 are database connection details to locate the definitions of the SQL object types.

JPublisher generates the following Java sources:

- Person.java is the Java class for the PERSON_T object.
- PersonRef.java is the Java class for a REF to a PERSON_T object.
- PersonArray.java is a collection of Person objects.

If the command line in Listing 11.25 had included the "–methods=true" option, a Person.sqlj file would be generated instead of the Person.java file. In addition, the Person.sqlj file would include Java wrapper methods for each method defined in the PERSON_T object type. The names generated for wrapper methods can be changed by using an SQLJ Translator input file with appropriate TRANSLATE commands.

The generated wrapper methods invoke the SQL object methods by executing a PL/SQL anonymous block in a SQLJ statement. However, now focus on how you can use the generated classes to perform SQLJ operations from Java on the collections. Listing 11.26 shows the resulting method signatures for the PersonArray.java source that can be used to manage a collection of objects.

```
01: public class PersonArray
02:   implements CustomDatum, CustomDatumFactory
03: {
04:   public static CustomDatumFactory getFactory();
05:   public PersonArray();
06:   public PersonArray(Person[] a);
07:
08:   /* CustomDatum interface */
09:   public Datum toDatum(OracleConnection c) …;
10:   /* CustomDatumFactory interface */
11:   public CustomDatum create(Datum d, int sqlType) …;
12:
13:   public int length();
14:   public int getBaseType();
15:   public String getBaseTypeName();
16:   public ArrayDescriptor getDescriptor();
17:
18:   /* array accessor methods */
19:   public Person[] getArray();
20:   public void setArray(Person[] a);
21:   public Person[] getArray(long index, int count);
22:   public void setArray(Person[] a, long index);
23:   public Person getElement(long index);
24:   public void setElement(Person a, long index);
25: }
```

LISTING 11.26 Java object method signatures for the SQL collection

Notes on Listing 11.26:

- Lines 4– 6 are the methods used to construct the array object.
- Lines 8–11 are the methods required because of the implementing of the CustomDatum and CustomeDatumFactory interfaces.
- Lines 13–24 are methods for accessing information about the collection, including getting a specific Person element and adding new Person elements.

The collection can be read and written using SQLJ statements, and the above methods are useful for constructing and using the collection in your application.

11.5.3 ACCESSING THE SQL COLLECTIONS FROM JAVA

The example in Listing 11.27 shows how to construct a Java array of Person objects and add it to a PersonArray object, which, in turn, is inserted as a new record into the family table (see Listing 11.24b), and then shows how to read the contents of the family table.

```
01: import sqlj.runtime.*;
02: import sqlj.runtime.ref.*;
03:
04: public class VarArrayExample {
05:
06:   #sql iterator Families (int id, PersonArray members);
07:
08:   public VarArrayExample() {
09:     DefaultContext ctx = null;
10:     Families family = null;   // declare iterator variable
11:     String driver = "jdbc:oracle:thin:";
12:     try {
13:       Class.forName("oracle.jdbc.driver.OracleDriver");
14:       ctx = new DefaultContext(
15:          driver + "bookstore/bookstore@localhost:1521:ORA815",
16:          false);
17:       DefaultContext.setDefaultContext(ctx);
18:
19:       String[] firstNames = { "Jackie", "Larry", "Sandra" };
20:       String[] lastNames = { "Chandra", "Chandra", "Chandra" };
21:       /*
22:       ** Construct the array of Person objects built from
23:       ** the array of first and last names. Generate
```

LISTING 11.27 Accessing SQL varying-array collection in SQLJ

```
24:       ** a new id for each person, and a date of birth
25:       ** based on the loop iteration variable value
26:       */
27:       Person[] members = new Person[firstNames.length];
28:       for (int i = 0; i < members.length; i++) {
29:         members[i] = new Person();
30:         members[i].setId(new java.math.BigDecimal(i+20));
31:         members[i].setFirstname(firstNames[i]);
32:         members[i].setLastname(lastNames[i]);
33:         members[i].setBirthdate(
34:           new java.sql.Timestamp(72+i, 1+i, 10+i, 0, 0, 0, 0));
35:       }
36:
37:       /*
38:       ** COnstruct the PersonArray collection from the
39:       ** array Person objects in the member variable
40:       */
41:       PersonArray aFamily = new PersonArray(members);
42:       /*
43:       ** Use an SQLJ Insert to create a new family, assigning
44:       ** an unique id for the family generated from an
45:       ** Oracle sequence called family_seq.
46:       */
47:       #sql { insert into family (id, members)
48:              values (family_seq.nextval, :aFamily) };
49:
50:       /*
51:       ** Now populate the family iterator with
52:       ** each family record, where the members
53:       ** column is read as a PersonArray object
54:       */
55:       #sql family = { select id, members from family };
56:       while (family.next()) {
57:         PersonArray pa = family.members();
58:         System.out.println("Family: " + family.id());
59:
60:         for (int j = 0; j < pa.length(); j++) {
61:           Person p = pa.getElement(j);
62:           System.out.println("\t" + p.getId() + " " +
63:                              p.getFirstname() + " " +
64:                              p.getLastname() + " " +
65:                              p.getBirthdate());
66:         }
```

LISTING 11.27 *Continued*

```
67:       }
68:       family.close();
69:       #sql { commit; };
70:       ctx.close();
71:     }
72:     catch (Exception e) {
73:       e.printStackTrace();
74:     }
75:   }
76:
77:   public static void main(String[] args) {
78:     new VarArrayExample();
79:   }
80: }
```

LISTING 11.27 *Continued*

Notes on Listing 11.27:

- Line 6 creates a named iterator class, called Families, which is used to query the SQL varying-array collection column. The second iterator data type uses the PersonArray class to receive the SQL collection for the member column in the FAMILY table.
- Line 10 declares the iterator variable called families used for querying the SQL collection data.
- Lines 27–35 create the array of Person objects to be used for the insert operation.
- Line 41 instantiates the PersonArray collection class with elements from the member array of Person objects. The PersonArray construct accepts an array of Person objects (see line 6 of Listing 11.26).
- Lines 47 and 48 show a simple SQL INSERT statement that accepts the PersonArray collection object referenced by the aFamily variable. This is all that is required to insert a new collection of varying-array objects.
- Line 55 issues the query on the family table and returns the result set to the "families" iterator variable.
- Line 56 is the start of the iterator loop mechanism that calls the next() method to step through each family record retrieved from the database table.
- Line 57 uses the named iterators members() method to return a PersonArray object reference to the SQL varying-array collection for a specific family record.

- Lines 60–66 show an inner loop using the PersonArray length() method to control the loop. It calls the getElement() method to obtain a reference to each Person object contained in the PersonArray object, and prints some of the details of each Person object in the family record.

Listing 11.27 shows how the use of SQLJ and JPublisher technology can simplify accessing and manipulating collections of SQL objects.

Using JPublisher, a class called PersonNestedTable is generated for the PERSON_NT_T nested table. Listing 11.28 shows an example using the Person and PersonNestedTable classes on data contained in the TEAM table, which contains a nested table of PERSON_T objects for the team members (see Listing 11.24b).

```
01: import sqlj.runtime.*;
02: import sqlj.runtime.ref.*;
03: import java.io.*;
04: import java.util.*;
05: import java.math.*;
06:
07: public class NestedTableExample {
08:
09:   String driver = "jdbc:oracle:thin:";
10:   String url = driver +
11:                "bookstore/bookstore@localhost:1521:ORA815";
12:
13:   #sql iterator TeamMember (int id, PersonNestedTable members);
14:
15:   public NestedTableExample(int teamId, String fileName) {
16:     DefaultContext ctx = null;
17:     TeamMember team = null;
18:
19:     try {
20:       Class.forName("oracle.jdbc.driver.OracleDriver");
21:       ctx = new DefaultContext(url, false);
22:       DefaultContext.setDefaultContext(ctx);
23:
24:       Vector memberList = readMembers(fileName);
25:
26:       PersonNestedTable newTeam = new PersonNestedTable(
27:                     new Person[memberList.size()]);
28:       for (int i = 0; i < memberList.size(); i++) {
29:         newTeam.setElement((Person) memberList.elementAt(i), i);
30:       }
```

LISTING 11.28 Accessing a SQL nested-table collection

```
31:
32:       #sql { insert into team (id, members)
33:              values (:teamId, :newTeam) };
34:       System.out.println("New Team inserted");
35:
36:       #sql team = { select id, members from team };
37:       while (team.next()) {
38:         PersonNestedTable pa = team.members();
39:
40:         System.out.println("Team: " + team.id());
41:         for (int j = 0; j < pa.length(); j++) {
42:           Person p = pa.getElement(j);
43:           System.out.println("\t" + p.getId() + " " +
44:             p.getFirstname() + " " + p.getLastname() + " " +
45:             p.getBirthdate());
46:         }
47:       }
48:       team.close();
49:       ctx.close();
50:     }
51:     catch (Exception e) {
52:       e.printStackTrace();
53:     }
54:   }
```

LISTING 11.28 *Continued*

Notes on Listing 11.28:

- Line 13 declares a SQLJ iterator for reading the rows from the TEAM table, whose second column is a nested table.
- Line 17 declares the iterator variable for the query result set.
- Line 24 calls a readMembers() method (see Listing 11.29) to read member records from a text file to build a vector of members. This step is needed in order to find out how many elements are needed to size the array of Person objects instantiated in the argument for the PersonNestedTable constructor in lines 26 and 27.
- Lines 26 and 27 create the PersonNestedTable collection object, which must have an array argument whose size is pre-allocated before you can set each array element to contain a Person object.
- Lines 28–30 copy the Person object references from the vector into the nested-table collection used in the insert operation.

- Lines 32 and 33 execute the SQLJ INSERT statement to add a new TEAM row with a collection of members.
- Line 36 initiates a query to process the contents of the TEAM table, returning a result set to the team iterator variable (declared in line 17).
- Line 38 obtains a reference to a team row nested-collection object.
- Lines 41–46 loop through each of the nested-table elements to print the Person object contents.

It is interesting that the SQL varying-array and nested-table collections can both be read into either a PersonArray or PersonNestedTable object. This is only possible because the PersonArray and PersonNestedTable classes are structurally similar with similar method calls. However, you should use classes appropriate for the related database type.

```
56:   public Vector readMembers (String fileName) throws Exception {
57:       BufferedReader br = new BufferedReader(
58:                                new FileReader(fileName));
59:       Vector memberList = new Vector();
60:       StringTokenizer st = null;
61:       Person member = null;
62:       int idx = 0;
63:       String str = null;
64:
65:       while ((str = br.readLine()) != null) {
66:          member = new Person();
67:          st = new StringTokenizer(str, ":", false);
68:          while (st.hasMoreTokens()) {
69:            int id = 0;
70:            try { id = Integer.parseInt(st.nextToken()); }
71:            catch (Exception nfe) { id = 1; }
72:            member.setId(new BigDecimal(id));
73:            member.setFirstname(st.nextToken());
74:            member.setLastname(st.nextToken());
75:            member.setBirthdate(
76:              new java.sql.Timestamp(72+28, 03, id, 0, 0, 0, 0));
77:            memberList.addElement(member);
78:          }
79:          idx++;
80:       }
```

LISTING 11.29 Creating a collection of members from a text file

```
81:       br.close();
82:       return memberList;
83:   }
84:
85:   public static void main(String[] args) {
86:     new NestedTableExample(2, "team1.txt");
87:   }
88: }
```

LISTING 11.29 *Continued*

The remaining piece of code for this example is shown in Listing 11.29. It reads a text file of tokenized member information.

The readMembers() method reads the member data on each line in the file. Each line in the file has the member id, first name, and last name separated by a colon. The java.util.StringTokenizer object is used to extract the field values for a member to build each Person object. The Person objects are added to a Java vector. The vector is returned to the caller of the readMembers() method as a collection of Person objects.

11.6 MANAGING LARGE DATA TYPES

In Chapter 10, you learned how to read LONG or large-object LOB columns. In this section you will look at similar examples using SQLJ. The SQLJ runtime class library provides three classes for working with large objects:

- AsciiStream—for processing character data in bytes.
- BinaryStream—for processing binary data in bytes.
- UnicodeStream—for processing character data in 16-bit characters.

All of these classes, which are defined in the sqlj.runtime package, are subclasses of sqlj.runtime.StreamWrapper. The StreamWrapper class is a subclass of java.io.FilterInputStream. These classes act as an input source for data inserted into large columns, or an input source when extracting data from large columns.

The key thing to remember when processing streams associated with database columns is that you must process their contents before you work with another column, and before you move to the next row. Positional iterators always declare the stream column last, and are limited to only one stream object per query. Named iterators do not have this restriction. SQLJ imposes the additional restriction that you cannot use a stream object in the INTO clause of a SELECT statement. Therefore, most of the code examples in this section make use of iterators, except for the examples that operate on the CLOB, BLOB, and BFILE locators.

The examples that follow show how to use stream classes in SQLJ to read from, and write to, a database column. The examples are presented as the method only, which you add to any class.

11.6.1 READING FROM A LONG COLUMN

```
01: #sql iterator LongAscii (int getId, int getLen,
02:                          AsciiStream getData);
03: public void readLongAscii(int idVal) {
04:   try {
05:     LongAscii iter;
06:     #sql iter = { select id getid, len getlen, data getdata
07:                   from demo_long
08:                   where id = :idVal};
09:     while (iter.next()) {
10:       System.out.println("Record id: " + iter.getId());
11:       byte[] buf = new byte[iter.getLen()];
12:       AsciiStream aStream = iter.getData();
13:       aStream.read(buf);
14:       aStream.close();
15:       StringBuffer sb = new StringBuffer(buf.length);
16:       for (int i = 0; i < buf.length; i++) {
17:          sb.append((char)buf[i]);
18:       }
19:       System.out.println(sb);
20:     }
21:     iter.close();
22:   }
23:   catch (Exception e) {
24:     e.printStackTrace();
25:   }
26: }
```

The key steps in reading text from a LONG column are:

1. Get the column as an AsciiStream object (line 12 gets the stream from the iterator column).
2. Read the data from the stream (line 13).
3. Close the stream (line 14).

The remainder of the example adds the byte array to a StringBuffer for printing on the screen.

11.6.2 WRITING TO A LONG COLUMN

```
01: public void writeLongAscii(int nextId, String filename) {
02:   try {
03:     File f = new File(filename);
04:     if (f.exists()) {
05:       int len = (int) f.length();
06:       AsciiStream inData = new AsciiStream(
07:                                  new FileInputStream(f), len);
08:       #sql { insert into demo_long (id, len, data)
09:              values (:nextId, :len, :inData) };
10:       inData.close();
11:       #sql { commit };
12:     }
13:     else {
14:        System.out.println("File " + f.getAbsolutePath() +
15:                           " does not exist");
16:     }
17:   }
18:   catch (Exception e) {
19:     e.printStackTrace();
20:   }
21: }
```

Writing a LONG column requires you to associate an input stream with an AsciiStream object. The example uses a file as the input source and stores the contents of the file in the LONG column.

- Lines 6 and 7 use the AsciiStream constructor with two arguments, the input stream, and the length of the data. The length is very important for the example to work. Alternatively, you can construct the stream object using a new AsciiStream (InputStream), and call the setLength() method to ensure that the data volume is set before executing the insert statement.
- Lines 8 and 9 perform the insert, and the AsciiStream is processed as a bind variable.
- Line 10 closes the input stream.

11.6.3 READING FROM A LONG RAW COLUMN

```
01: #sql iterator LongRaw (int getId, int getLen,
02:                        BinaryStream getData);
03: public void readLongRaw(int idVal) {
04:   try {
05:     LongRaw iter;
06:     #sql iter = { select id getid, len getlen, data getdata
```

```
07:                   from demo_longraw
08:                   where id = :idVal};
09:     while (iter.next()) {
10:       int len = (int) iter.getLen();
11:       System.out.println("Record id: " + iter.getId());
12:       BinaryStream aStream = iter.getData();
13:       byte[] buf = new byte[len];
14:       aStream.read(buf);
15:       aStream.close();
16:       PictureFrame pic = new PictureFrame(buf);
17:       pic.setVisible(true);
18:     }
19:     iter.close();
20:   }
21:   catch (Exception e) {
22:     e.printStackTrace();
23:   }
24: }
```

Reading from a LONG RAW column follows the same steps as reading from a LONG column. However, in this case, a BinaryStream is used, as shown in bold. The same PictureFrame class that is described in Chapter 10 is used to display the image in a Java frame. Here is a sample of what an image would look like using the PictureFrame class:

11.6.4 WRITING TO A LONG RAW COLUMN

```
01: public void writeLongRaw(int nextId, String filename) {
02:   try {
03:     File f = new File(filename);
04:     if (f.exists()) {
05:       int len = (int) f.length();
06:
07:       BinaryStream inData = new BinaryStream(
08:                                 new FileInputStream(f), len);
09:       #sql { insert into demo_longraw (id, len, data)
10:              values (:nextId, :len, :inData) };
11:       inData.close();
12:       #sql { commit };
13:     }
14:     else {
15:       System.out.println("File " +
16:              f.getAbsolutePath() + " does not exist");
17:     }
18:   }
19:   catch (Exception e) {
20:     e.printStackTrace();
21:   }
22: }
```

Writing a file to a LONG RAW column is similar to writing to a LONG column, but uses a BinaryStream object, as shown in bold.

11.6.5 READING FROM A CLOB

```
01: public void readClob(int idVal) {
02:   try {
03:     oracle.sql.CLOB theClob = null;   // not portable
04:     #sql { select cdata into :theClob
05:            from demo_clob
06:            where id = :idVal};
07:     BufferedReader bf = new BufferedReader(
08:                           theClob.getCharacterStream());
09:     System.out.println("Length of data: " + theClob.length());
10:     String s;
11:     while ((s = bf.readLine()) != null) {
12:       System.out.println(s);
13:     }
14:     bf.close();
15:   }
16:   catch (Exception e) {
```

```
17:     e.printStackTrace();
18:   }
19: }
```

Reading from a CLOB column is straightforward, as seen in the preceding example. The bold text shows the declaration of the oracle.sql.CLOB object in line 3.[14] The SQLJ SELECT statement reads the CLOB locator into the CLOB object. Line 8 obtains a stream object from the CLOB locator, and the contents are accessed as you would a standard Java stream.

11.6.6 WRITING TO A CLOB

```
01: public void writeClob(int newId, String filename) {
02:   try {
03:     oracle.sql.CLOB theClob = null;   // not portable
04:
05:     #sql { begin
06:              insert into demo_clob (id, cdata)
07:              values (:newId, empty_clob())
08:              returning cdata into :out theClob;
09:            end;
10:     };
11:     if (theClob != null) {
12:       PrintWriter out = new PrintWriter(
13:                           theClob.getCharacterOutputStream());
14:       BufferedReader in = new BufferedReader(
15:                             new FileReader(filename));
16:       String s;
17:       while ((s = in.readLine()) != null) {
18:         out.println(s);
19:       }
20:       in.close();
21:       out.close();
22:     }
23:     else {
24:       System.out.println("The clob is null");
25:     }
26:   }
27:   catch (Exception e) {
28:     e.printStackTrace();
29:   }
30: }
```

[14]The SQLJ Translator issues a warning to indicate that the oracle.sql.CLOB column is not portable. When the Oracle SQLJ Translator is updated to support the JDBC 2.0 CLOB class, portability will be possible.

Writing to a CLOB column requires two major steps:

1. Create the CLOB locator value using the Oracle database built-in function EMPTY_CLOB().
2. Write the CLOB contents, after getting an output stream from the CLOB locator.

In the example, lines 5–10 execute a PL/SQL anonymous block that creates the LOB locator, and returns the value to the Java application. If you are using a database prior to Oracle8i, then you have to:

- ❑ Execute the INSERT statement to create the empty LOB.
- ❑ Execute a SELECT statement to retrieve the LOB locator value.
- ❑ Write to the LOB using a stream object.

The PL/SQL anonymous block was used because it allows the use of the DML returning clause, and combines the two SQL steps of creating the LOB locator and reading it into one operation.

11.6.7 READING FROM A BLOB

```
01: public void readBlob(int idVal) {
02:   try {
03:     oracle.sql.BLOB theBlob = null;   // not portable
04:     #sql { select bdata into :theBlob
05:            from demo_blob
06:            where id = :idVal};
07:     InputStream in = theBlob.getBinaryStream();
08:     System.out.println("Length of data: " + theBlob.length());
09:     byte[] buf = new byte[(int)theBlob.length()];
10:     in.read(buf);
11:     in.close();
12:     PictureFrame pic = new PictureFrame(buf);
13:     pic.setVisible(true);
14:   }
15:   catch (Exception e) {
16:     e.printStackTrace();
17:   }
18: }
```

Reading from a BLOB column requires that you obtain the BLOB locator column and then a binary stream object. Once the binary stream object is created, you process the data using the stream methods. The example assumes that it is reading an image object, which is then passed to the PictureFrame class.

11.6.8 WRITING TO A BLOB

```
01: public void writeBlob(int newId, String filename) {
02:   try {
03:     oracle.sql.BLOB theBlob = null;   // not portable
04:
05:     #sql { begin
06:              insert into demo_blob (id, bdata)
07:              values (:newId, empty_blob())
08:              returning bdata into :out theBlob;
09:            end;
10:     };
11:     if (theBlob != null) {
12:       OutputStream out = theBlob.getBinaryOutputStream();
13:       InputStream in = new FileInputStream(filename);
14:       int dataByte = 0;
15:       while ((dataByte = in.read()) != -1) {
16:         out.write(dataByte);
17:       }
18:       in.close();
19:       out.close();
20:     }
21:     else {
22:       System.out.println("The blob is null");
23:     }
24:   }
25:   catch (Exception e) {
26:     e.printStackTrace();
27:   }
28: }
```

Writing to a BLOB also requires that you first obtain the LOB locator, and then route the binary output stream to the BLOB contents. Using the write() methods of the output stream, you can modify the contents of the LOB.

11.6.9 READING FROM A LONG COLUMN WITH A UNICODESTREAM

```
01: #sql iterator LongUnicode (int getId, int getLen,
02:                            UnicodeStream getData);
03: public void readLongUnicode(int idVal) {
04:   try {
05:     LongUnicode iter;
06:     #sql iter = { select id getid, len getlen, data getdata
07:                   from demo_long
08:                   where id = :idVal};
09:     while (iter.next()) {
```

```
10:       int len = iter.getLen();
11:       System.out.println("Record: " + iter.getId());
12:       UnicodeStream aStream = iter.getData();
13:       StringBuffer sb = new StringBuffer(len);
14:       for (int i = 0; i < len; i++) {
15:          sb.append((char)aStream.read());
16:       }
17:       aStream.close();
18:       System.out.println(sb);
19:     }
20:     iter.close();
21:   }
22:   catch (Exception e) {
23:     e.printStackTrace();
24:   }
25: }
```

The example here uses a UnicodeStream to read 16-bit characters from a LONG column. The iterator reads the LONG column as a UnicodeStream, and the stream is accessed using the read() methods. The cast to (char), in line 15, is done to add a Unicode character to the StringBuffer.

11.6.10 WRITING TO A LONG COLUMN WITH A UNICODESTREAM

```
01: public void writeLongUnicode(int newId, String filename) {
02:   try {
03:     File f = new File(filename);
04:     if (f.exists()) {
05:       int len = (int) (f.length() / 2);
06:       UnicodeStream inData = new UnicodeStream(
07:                                  new FileInputStream(f), len);
08:       #sql { insert into demo_long (id, len, data)
09:              values (:nextId, :len, :inData) };
10:       inData.close();
11:       #sql { commit };
12:     }
13:     else {
14:        System.out.println("File " +
15:          f.getAbsolutePath() + " does not exist");
16:     }
17:   }
18:   catch (Exception e) {
19:     e.printStackTrace();
20:    }
21: }
```

Writing a Unicode file to LONG column requires the following steps:

1. Determine the length of the file in characters by calling the length() method to obtain its length in bytes and then dividing by two, as shown in line 5.
2. Create the UnicodeStream object using the appropriate constructor, and ensure that the correct length is set, as shown in lines 6 and 7.
3. Execute the INSERT statement, passing the Unicode stream object as a bind value. The Oracle JDBC driver does the rest.

Reading or writing Unicode characters with the UnicodeStream object requires that the high byte be first and the low byte second.[15]

11.6.11 READING A BFILE

To read a BFILE, you first obtain the locator, and then use it like a file handle to open the file and read the contents.

```
01: public void readBFile(int idVal) {
02:   try {
03:     oracle.sql.BFILE bfile = null;
04:
05:     #sql { select fileptr into :bfile
06:            from demo_bfile
07:            where id = :idVal };
08:
09:     bfile.openFile();
10:     byte[] buf = bfile.getBytes(1L, (int)bfile. length());
11:     bfile.closeFile();
12:     PictureFrame pic = new PictureFrame(buf);
13:     pic.setVisible(true);
14:   }
15:   catch (Exception e) {
16:     e.printStackTrace();
17:   }
18: }
```

The example uses the openFile() method of the BFILE object before reading the contents. The length() method is used to get the length of the BFILE in bytes.

Alternatively, you can read the contents of the BFILE using the following code:

```
InputStream in = bfile.getBinaryStream();
byte[] buf = new byte[(int) bfile.length()];
```

[15]On the Microsoft Windows NT platform, Notepad creates Unicode files with the low byte first. Therefore, you need to byte swap each character read from the file.

```
        in.read(buf);
        in.close();
```

In this alternative example, you request a stream object (in this case a binary stream), and then read the contents using the stream read() method.

11.6.12 WRITING A BFILE

Writing a BFILE is simply the act of inserting a logical link from the database to the external file.

```
01: public void writeBfile(int newId, String filename) {
02:   try {
03:     #sql { insert into demo_bfile (id, fileptr)
04:       values (:newId, bfilename('BFILE_DIR', :filename)) };
05:   }
06:   catch (Exception e) {
07:     e.printStackTrace();
08:   }
09: }
```

The code example inserts a new record into a table that holds the BFILE locators that reference the external files. This is accomplished by calling the Oracle RDBMS built-in function called BFILENAME. The first argument to the BFILENAME is an Oracle8 DIRECTORY object, and the second parameter is the name of the file located in the directory. The directory object is created with the CREATE DIRECTORY statement, which defines a logical name and association for a physical directory in the operating system. The hard-coded directory name should be replaced with a parameterized value.

11.7 EXECUTING STORED PROCEDURES AND FUNCTIONS

As in JDBC, the SQLJ environment allows you to execute stored procedures in a vendor-independent way, regardless of the language used to write the stored procedure. In the Oracle RDBMS environment, the SQLJ Translator also allows you to invoke PL/SQL anonymous blocks, as seen in the CLOB and BLOB examples in the preceding section. Here the focus is on the syntactic aspects of calling a procedure or a function, without specific examples.

11.7.1 CALLING A STORED PROCEDURE

The syntax used to call a stored procedure is:

```
#sql { call procedure-name [(arguments, …)] };
```

The procedure name can include an Oracle database schema name and a package name, using the standard dot notation to qualify the procedure.

The arguments are optional, depending on the formal parameters declared in the procedure call. When using Oracle7 databases, you must omit the brackets if there are no arguments.

11.7.2 CALLING A STORED FUNCTION

Calling a stored function requires the following syntax:

```
type-name result;

#sql result = { values (function-name [(arguments, …)]) };
```

In the syntax shown, the type-name is the return data type expected for the function return value. The VALUES keyword is required, and the called function name is placed inside brackets. The function name can include an Oracle database schema name and/or a PL/SQL package name, and have optional arguments. The function result is stored in the Java variable whose name is listed before the assignment operator.[16]

11.7.3 STORED PROCEDURE OR FUNCTION ARGUMENTS

Stored procedures and functions accept arguments using different parameter-passing methods. In the Oracle environment, a parameter has one of the following modes:

- IN—accepts a value from an input-only argument.
- OUT—returns a value to the caller using an output-only argument.
- INOUT—accepts a value from the caller, and returns a modified value using the same argument.

When you call a stored procedure/function in SQLJ, you must explicitly identify the mode used for each bind variable used as a parameter. The syntax for specifying a parameter-passing mode is to include one of the keywords, IN, OUT, or INOUT, immediately after the colon and before the bind variable name. For example:

```
int id;
int custName;

#sql { get_customer(:in id, :out custName) };
```

[16]The function result does not need a preceding colon because it is outside the curly braces. The general rule is: if the Java variable is inside the curly braces, then it must be preceded by a colon to be treated as a bind variable.

The example calls a procedure called get_customer, passing an input integer argument as the first parameter, and receives the customer name from the second output string argument.

SUMMARY

SQLJ is a standard way to embed SQL statements in Java code in order to interact with a relational database. The simplicity of using SQLJ has been demonstrated; its coding benefits include:

- Reducing the amount of code to be written.
- Stronger type checking and validation at translation time.

You have read about using SQLJ to perform most of the same tasks you can do in JDBC, such as executing SQL statements, stored procedures, and functions, and processing large object data types and complex structures like SQL objects. You were introduced to the Oracle JPublisher utility that showed how to generate a Java class for an SQL object type that allows your SQLJ applications to work with database SQL object data in a way natural to a Java developer.

Using SQLJ, or JDBC, a set of classes can be developed that encapsulate the logic needed to access the database. An application developer can focus on building the business process logic to use the classes that provide database access. Subsequent changes to the data-access implementation classes can minimize the impact on changes made to the application business-process logic. The task of writing a class to manage the business rules that manage the data is time-consuming. SQLJ can reduce the time it takes to develop the database class library.

As an alternative, you can use Oracle JDeveloper to generate a set of classes that conform to a framework called Business Components for Java that encapsulates the data-access layer and associated data-validation rules. This is the next step toward even more rapid application development for your enterprise class applications. If you do not use Oracle JDeveloper, you have to handcraft the framework or the data access layer API yourself.