CHAPTER 6

SQLJ: Embedded SQL in Java

This chapter introduces SQLJ, a relatively new standard that many database vendors have already adopted. SQLJ differs from JDBC in that SQL statements can be embedded directly in Java programs and translation-time semantics checks can be performed. It is a powerful tool that complements JDBC access to databases. Programming in SQLJ is discussed in detail in this chapter, and an application program for the investment portfolio database is presented.

6.1 What Is SQLJ?

SQLJ is an emerging database programming tool that allows embedding of static SQL statements in Java programs, very much like Pro*C or Pro*C++. SQLJ is an attractive alternative to JDBC because it allows translation-time syntax and semantics checking of static SQL statements. As a consequence, application programs developed in SQLJ are more robust. SQLJ's syntax is also much more compact than that of JDBC, resulting in shorter programs and increased user productivity.

The SQLJ translator converts Java programs embedded with static SQL statements into pure Java code, which can then be executed through a JDBC driver against the database. Programmers can also perform dynamic SQL access to the database using JDBC features.

6.2 Simple Example

A simple program in SQLJ is presented in this section. This program illustrates the essential steps that are needed to write an SQLJ program. These steps follow:

1. *Import necessary classes.* In addition to the JDBC classes, `java.sql.*`, every SQLJ program will need to include the SQLJ run-time classes `sqlj.runtime.*` and `sqlj.runtime.ref.*`. In addition, to establish the default connection to Oracle, the `Oracle` class from the `oracle.sqlj.runtime.*` package is required. So, a typical set of statements to import packages would be:

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.io.*;
import oracle.sqlj.runtime.*;
```

2. *Register the JDBC driver, if needed.* If a non-Oracle JDBC driver is being used, a call to the `registerDriver` method of the `DriverManager` class is necessary. For the purposes of this chapter, an Oracle JDBC driver is assumed. Therefore, this statement is not shown in any of the examples.

3. *Connect to the database.* Connecting to the Oracle database is done by first obtaining a `DefaultContext` object using the `getConnection` method (of the `Oracle` class[1]), whose specification is

```
public static DefaultContext getConnection
   (String url,String user,String password,boolean autoCommit)
throws SQLException
```

   `url` is the database URL, and `user` and `password` are the Oracle user ID and password, respectively. Setting `autoCommit` to `true` would create the connection in autocommit mode, and setting it to `false` would create a connection in which the transactions must be committed by the programmer. A sample invocation is shown here:

```
DefaultContext cx1 =
  Oracle.getConnection("jdbc:oracle:oci8:@",
                       "book","book",true);
```

1. The `Oracle` class can be found in the package `oracle.sqlj.runtime.*`.

The `DefaultContext` object so obtained is then used to set the static default context, as follows:

```
DefaultContext.setDefaultContext(cx1);
```

This `DefaultContext` object now provides the default connection to the database.

4. *Embed SQL statements in the Java program.* Once the default connection has been established, SQL statements can be embedded within the Java program using the following syntax:

   ```
   #sql {<sql-statement>}
   ```

   where `#sql` indicates to the SQLJ translator, called `sqlj`, that what follows is an SQL statement and `<sql-statement>` is any valid SQL statement, which may include *host variables* and *host expressions*. Host variables are prefixed with a colon, much like in Pro*C/Pro*C++.

The following simple SQLJ program performs a query against the investment portfolio database. It reads a security symbol from the user and performs a simple query to retrieve information about the particular security. The program uses the `readEntry` method presented in Chapter 5.

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import java.io.*;
import oracle.sqlj.runtime.*;

public class Simple1 {
  public static void main (String args[]) throws SQLException {

    DefaultContext cx1 =
      Oracle.getConnection("jdbc:oracle:oci8:@",
                           "book","book",true);
    DefaultContext.setDefaultContext(cx1);

    String cn;
    Double ap,bp,cp;
    String sym = readEntry("Enter symbol : ").toUpperCase();
```

```
        try {
          #sql {select cname,current_price,ask_price,bid_price
                 into   :cn,:cp,:ap,:bp
                 from   security
                 where  symbol = :sym };
        } catch (SQLException e) {
            System.out.println("Invalid symbol.");
            return;
          }
        System.out.println("\n  Company Name  = " + cn);
        System.out.println("  Last sale at  = " + cp);
        if (ap == null)
          System.out.println("  Ask price     = null");
        else
          System.out.println("  Ask price     = " + ap);
        if (bp == null)
          System.out.println("  Bid price     = null");
        else
          System.out.println("  Bid price     = " + bp);
      }
    }
```

The program uses several Java variables (host variables) in the SQL query. It also checks to see if any of the values returned from the database are `null`s. A `null` value returned by the query is indicated by a Java `null` reference for the host variable into which the database value is retrieved. This feature of SQLJ implies that whenever there is a possibility of a `null` value being retrieved into a host variable, the host variable's Java type should not be a primitive type.

6.3 Compiling SQLJ Programs

The SQLJ translator takes as input an SQLJ program file (with suffix `.sqlj`) and produces a `.java` file along with several other SQLJ profile files that contain the classes necessary to perform the SQL operations. The translator also automatically invokes the Java compiler to produce a `.class` file.

There are several command-line parameters that can be given to the SQLJ translator. For example, if the users want online semantics checking of SQL statements, they can specify the following command-line options in compiling the `Simple1.sqlj` program:

```
% sqlj -url = jdbc:oracle:oci8:@ \
       -user = book -password = book Simple1.sqlj
```

Online semantics checking is performed by the SQLJ translator by connecting to the Oracle database using the user ID and password provided as command-line parameters.

One other commonly used command-line parameter is the `-warn` parameter. This parameter takes as its value a comma-separated list of options that either enables or disables certain warnings from being generated by `sqlj`. For example, the command

```
% sqlj -warn = noprecision,nonulls Simple1.sqlj
```

will disable warnings concerning loss of precision or possible retrieval of a `null` value into a Java primitive type.

SQLJ allows the possibility of providing these command-line parameters in a properties file. This is convenient when there are many command-line parameters that have to be specified. By default, the properties file is called `sqlj.properties`. This default can be overridden by specifying the properties file name in the `-props=` command-line option. A sample `sqlj.properties` file is

```
sqlj.driver = oracle.jdbc.driver.OracleDriver
sqlj.url = jdbc:oracle:oci8:@
sqlj.user = book
sqlj.password = book
sqlj.warn = noprecision,nonulls
```

The options mentioned in this sample file are the JDBC driver name, the connect string URL, the Oracle user and password, and the warnings flags.

6.4 Multiple Connections

Application programs written in SQLJ can easily access data from several databases by creating one `DefaultContext` object for the default database connection and one nondefault connection context for each additional database connection that is required. A nondefault connection context class called `DbList` is declared as follows:

```
#sql context DbList;
```

This declaration is expanded by `sqlj` into a Java class called `DbList`, which can then be instantiated in the SQLJ program as follows:

```
DbList x2 = new DbList(Oracle.getConnection(
    "jdbc:oracle:oci8:@","book2","book2",true));
```

to create a new connection context. This connection context can then be used in embedded SQL statements as follows:

```
#sql [x2] {<sql-statement>};
```

The SQLJ translator also supports online SQL semantics checks on multiple connection contexts at translation time through command-line parameters that are optionally tagged with the connection context class name. For example, the `sqlj.properties` file for the previous multiple-connection scenario would be as follows:

```
sqlj.driver = oracle.jdbc.driver.OracleDriver
sqlj.warn = noprecision,nonulls
#
sqlj.url = jdbc:oracle:oci8:@
sqlj.user = book
sqlj.password = book
#
sqlj.url@DbList = jdbc:oracle:oci8:@
sqlj.user@DbList = book2
sqlj.password@DbList = book2
```

Any statements that are executed within the default connection context will be verified using the `book/book` schema connection, and any statements that are executed within the `DbList` connection context will be verified using the `book2/book2` schema connection.

The following SQLJ program illustrates multiple connections:

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import java.io.*;
import oracle.sqlj.runtime.Oracle;

#sql context Book2Context;

public class Simple2 {
  public static void main (String args[])
      throws SQLException {
    DefaultContext x1 = Oracle.getConnection(
             "jdbc:oracle:oci8:@","book","book",true);
    DefaultContext.setDefaultContext(x1);
```

```
    Book2Context x2 = new Book2Context(
              Oracle.getConnection(
             "jdbc:oracle:oci8:@","book2","book2",true));

    String dbname="";
    try {
      #sql [x2] { select db_name
                  into   :dbname
                  from   db_list
                  where  db_name like 'C%' };
    } catch (SQLException e) {
        System.out.println("Error:" + e.getMessage());
        return;
      }
    System.out.println("DB name is " + dbname);

    String cn = "";
    String sym =
             readEntry("Enter symbol : ").toUpperCase();
    try {
      #sql { select cname
             into   :cn
             from   security
             where  symbol = :sym };
    } catch (SQLException e) {
        System.out.println("Invalid symbol.");
        return;
      }
    System.out.println("\n  Company Name  = " + cn);
  }
}
```

In the preceding program, the `DefaultContext` object `x1` was designated as the default connection, and hence it was not necessary to include `x1` in the second query. It is assumed that a table called `db_list` with a column called `db_name` exists in the schema `book2/book2`. Note that the query would fail if the `db_list` table contained more than one row with the `db_name` value starting with the letter `C`. (An SQLJ iterator, introduced in Section 6.6, is necessary to process queries with multiple answers.)

6.5 Host Variables and Expressions

Host variables and expressions can be used in SQLJ to communicate values between the SQL statement and the Java environment. Host variables are either Java local variables, Java declared parameters, or Java class/instance variables. A host expression is any valid Java expression.

A host variable must be preceded by a colon (`:`) followed by `IN`, `OUT`, or `INOUT`,[2] depending on whether it is an input to the SQL statement, output to the SQL statement, or both. `IN` is the default for host variables and expressions (except in an `into` list) and may be left out. When using the `IN`, `OUT`, and `INOUT` tokens, the colon immediately precedes the token, and there must be a space between the token and the variable name. When not using the `IN` token for an input variable or the `OUT` token for an output variable, the colon can immediately precede the variable name. Two examples of host variables in SQL statements are

```
#sql {select a into :a1 from t where b = :b1};
#sql {select a into :OUT a1 from t where b = :IN b1};
```

In addition to host variables, Java host expressions such as arithmetic expressions, method calls with return values, instance or class variables, array elements, conditional expressions, logical expressions, and so on can be used in SQL statements. Complicated host expressions must appear within parentheses after the colon to ensure that they are interpreted properly by SQLJ. For example, the following embedded SQL statement updates the cash balance for a particular member to a value that is 100 more than the value of the variable `x`:

```
#sql {update member
      set    cash_balance = :(x+100)
      where  mid = :y };
```

At run time, the Java expressions are evaluated and then passed on to the SQL statement.

The host variables and expressions used in SQL statements must be compatible and be convertible to and from an SQL type. Figure 6.1 gives the mapping between commonly used Oracle data types and Java types.

The Java wrapper classes `Integer`, `Long`, `Float`, and `Double` should be used instead of their primitive counterparts when there is a possibility of a `null` value being communicated from or to the database. The use of Java wrapper classes is necessary because database `null` values are converted into Java `null` references

2. The `IN`, `OUT`, and `INOUT` tokens are not case sensitive.

Figure 6.1 Mapping between Oracle data types and Java types.

Oracle Type	Java Type
`number, number(n), integer, integer(n)`	`int, long`
`number, number(n), number(n,d)`	`float, double`
`char, varchar2`	`java.lang.String`
`date`	`java.sql.Date`
`cursor`	`java.sql.ResultSet` or SQLJ iterator objects

and vice versa, and it is not possible for Java primitive types to be assigned `null` references.

6.6 SQLJ Iterators

Processing query results, especially when they contain more than one row, requires SQLJ *iterators*. An SQLJ iterator is basically a strongly typed version of a JDBC result set and is associated with the underlying database cursor defined by an SQL query.

An SQLJ iterator declaration specifies a Java class that is automatically constructed by SQLJ. The iterator class contains instance variables and access methods that correspond to the column types and, optionally, column names of the SQL query associated with the iterator. When an SQLJ iterator is declared, programmers can specify either just the data types of the selected columns (*positional iterators*) or both the data types and column names of the selected columns (*named iterators*).

An iterator object is instantiated by executing an SQL query, and the SQL data that are retrieved into the iterator object are converted to Java types specified in the iterator declaration.

6.6.1 Named Iterators

A named iterator is declared by specifying both the data types and the column names that correspond to the select list items of the query. The syntax for declaring a named iterator is as follows:

```
#sql iterator iterName
      (colType1 colName1, ..., colTypeN colNameN);
```

where `iterName` is a name given to the iterator, each `colTypeI` is a valid Java type, and each `colNameI` is a select list item name in the query to be associated with this iterator. It is important that the names and types in the iterator match the names and types in the SQL query. The select list items do not have to be in the same order as they appear in the iterator, but each select list item must appear in the iterator.

As an example, consider the problem of printing a monthly report of all transactions for a particular member in the investment portfolio database. The SQL query to produce such a listing is

```
select to_char(trans_date,'DD-MON-YYYY') tdate,
       trans_type ttype, symbol, quantity
       price_per_share, commission, amount
from   transaction
where  mid = :mmid and
       to_char(trans_date, 'MM') = :month and
       to_char(trans_date, 'YYYY') = :year;
```

where `:mmid` is the host variable holding the member ID and `:month` and `:year` are the host variables for the month and year for which the listing is to be produced. The SQLJ iterator for this query is defined[3] as follows:

```
#sql iterator TReport(String tdate, String ttype,
                      String symbol, double quantity,
                      double price_per_share,
                      double commission, double amount);
```

Notice that the SQL query select list items and their data types match the column names and types mentioned in the iterator. SQLJ automatically creates a class, called `TReport`, for the iterator. This class has methods to access the values of each of the columns mentioned in the iterator.

Once the named iterator has been declared, it can be instantiated and populated in the Java program with the following statements:

```
TReport t = null;
#sql t =
  {select to_char(trans_date,'DD-MON-YYYY') tdate,
          trans_type ttype, symbol, quantity,
          price_per_share, commission, amount
   from   transaction
```

3. The iterator declaration is typically made in the Java source file for the application that uses it. However, since the iterator declaration defines a separate Java class, it must be declared outside of the application class.

```
    where  mid = :mmid and
           to_char(trans_date, 'MM') = :month and
           to_char(trans_date, 'YYYY') = :year };
```

The individual rows of the iterator can be accessed using the `next` method. Whenever `next` is called, it retrieves the next row from the iterator and returns `true`. If there is no next row, it returns `false`.

Once a row has been retrieved, the individual columns of the row can be accessed using the accessor methods that are automatically created by SQLJ. These accessor methods have the same names as the column names mentioned in the iterator declaration, which are also the names of the select list items.

The iterator is equipped with a `close` method, which should be called once the iterator has been processed.

The following is the `printReport` method, which prints a monthly report of transactions for a particular member. The member ID is an input parameter to this method. The method reads the month and year from the user and generates the monthly report of transactions.

```
private static void printReport(String mmid)
    throws SQLException, IOException {

  String mmid2;
  try {
    #sql { select distinct mid
           into   :mmid2
           from   transaction
           where  mid = :mmid };
  } catch (SQLException e) {
      System.out.println("No transactions");
      return;
    }
  String month = readEntry("Month(01 - 12): ");
  String year  = readEntry("Year(YYYY): ");

  TReport t = null;
  #sql t={select to_char(trans_date,'DD-MON-YYYY') tdate,
                trans_type ttype, symbol, quantity,
                price_per_share, commission, amount
          from   transaction
          where  mid = :mmid and
                 to_char(trans_date, 'MM') = :month and
                 to_char(trans_date, 'YYYY') = :year};
```

```
    // Print Report Header
    writeSpaces(15);
    System.out.println("MONTHLY TRANSACTION REPORT");
    writeSpaces(21);
    System.out.println(month + "/" + year);
    writeDashes(68); System.out.println();
    System.out.print("Date"); writeSpaces(9);
    System.out.print("Type"); writeSpaces(2);
    System.out.print("Symbol"); writeSpaces(5);
    System.out.print("Shares"); writeSpaces(5);
    System.out.print("PPS"); writeSpaces(4);
    System.out.print("Commission"); writeSpaces(2);
    System.out.println("Amount");
    writeDashes(68); System.out.println();

    while(t.next()) {
      System.out.print(t.tdate() + "  ");
      writeEntryRight(t.ttype(),6);
      writeSpaces(3);
      writeEntryLeft(t.symbol(),6);
      writeEntryRight(twoDigit(t.quantity()),10);
      writeEntryRight(twoDigit(t.price_per_share()),10);
      writeEntryRight(twoDigit(t.commission()),10);
      writeEntryRight(twoDigit(t.amount()),10);
      System.out.println();
    }
    writeDashes(68); System.out.println();
    t.close();
  }

  static String twoDigit(double f) {
    boolean neg = false;
    if (f < 0.0) {
      neg = true;
      f = -f;
    }
    long dollars = (long) f;
    int cents = (int) ((f - dollars) * 100);
    String result;
```

```
    if (cents <= 9)
      result = dollars + ".0" + cents;
    else
      result = dollars + "." + cents;
    if (neg)
      return "-" + result;
    else
      return result;
  }

  private static void writeEntryLeft(String text, int width) {
    System.out.print(text);
    for (int i = 0; i < (width - text.length()); i++)
        System.out.print(" ");
  }

  private static void writeEntryRight(String text, int width) {
    for (int i = 0; i < (width - text.length()); i++)
        System.out.print(" ");
    System.out.print(text);
  }

  private static void writeSpaces(int width) {
    for (int i = 0; i < width; i++)
      System.out.print(" ");
  }

  private static void writeDashes(int width) {
    for (int i = 0; i < width; i++)
      System.out.print("-");
  }
```

After the iterator has been populated using an SQL query, the `next` method is used to fetch the next row. For each row fetched, the accessor methods `t.tdate`, `t.symbol`, and so on are used to access the individual columns of the row.

The `printReport` method makes use of a dollar-formatting static method called `twoDigit`. It also uses other string-formatting methods including `writeSpaces`, `writeDashes`, `writeEntryLeft`, and `writeEntryRight`.

6.6.2 Positional Iterators

A positional iterator is declared in a manner similar to a named iterator, except that column names are not specified. The Java types into which the columns of the SQL query are retrieved must be compatible with the data types of the SQL data. The names of the SQL select list items are irrelevant.

Since the names of columns are not specified, the order in which the positional Java types are mentioned in the iterator must exactly match the order of the data types of the select list items of the SQL query.

The positional iterator is instantiated and populated in the same manner as a named iterator, but the manner in which the data are retrieved from the iterator is different. For a positional iterator, a `fetch into` statement is used along with a call to the `endFetch` method to determine if the last row has been reached. The syntax of the `fetch into` statement is as follows:

```
#sql { fetch :iter into :var1, ..., :vark };
```

where `iter` is the name of the positional iterator and `var1`, ..., `vark` are host variables of appropriate types that will receive values from the select list items. These variables must be in the same order as their corresponding select list items in the select list of the query.

The `endFetch` method, when applied to the iterator object, initially returns `true` before any rows have been fetched. It then returns `false` after each successful row fetch and finally returns `true` after the last row has been fetched. The call to `endFetch` must be done before the row is fetched, because the `fetch` does not throw an `SQLException` when trying to fetch after the last row.

As an example, consider the problem of printing the current, ask, and bid prices of a security, given a substring of the security name. The query to accomplish this task is

```
select symbol,cname,current_price,ask_price,bid_price
from   security
where  upper(cname) like :cn;
```

where `cn` is the substring of the security name. The positional iterator for this query is declared as follows:

```
#sql iterator PQuote(String,String,double,Double,Double);
```

Note that the Java data type corresponding to the `current_price` column is declared as `double` since this database column cannot contain `null`s.

The method to get the price quote for a security given a substring of the company name is

```
public static void getPriceQuoteByCname(String cn)
      throws SQLException, IOException {
  double cp=0.0;
  Double ap=null,bp=null;
  PQuote p = null;
  String sym = "";

  #sql p = {select symbol, cname, current_price,
                  ask_price, bid_price
           from   security
           where  upper(cname) like :cn};
  #sql {fetch :p into :sym,:cn,:cp,:ap,:bp};
  if (!p.endFetch()) {
    System.out.print("Symbol"); writeSpaces(12);
    System.out.print("Company"); writeSpaces(17);
    System.out.print("Last Sale"); writeSpaces(4);
    System.out.print("Ask"); writeSpaces(7);
    System.out.println("Bid");
    writeDashes(74);
    System.out.println();
    while (!p.endFetch()) {
      writeEntryLeft(sym,9);
      writeEntryLeft(cn,30);
      writeEntryRight(twoDigit(cp),10);
      if (ap == null)
        System.out.print("      null");
      else
        writeEntryRight(twoDigit(ap.doubleValue()),10);
      if (bp == null)
        System.out.print("      null");
      else
        writeEntryRight(twoDigit(bp.doubleValue()),10);
      System.out.println();
      #sql {fetch :p into :sym,:cn,:cp,:ap,:bp};
    };
    writeDashes(74);
  } else {
      System.out.println("No company matches the name");
    }
  p.close();
}
```

6.7 Dynamic SQL Using JDBC

SQLJ by nature caters to static SQL. However, there are situations when dynamic SQL is needed. The dynamic SQL API for SQLJ is JDBC; hence, an SQLJ program may contain both SQLJ and JDBC code. Access to JDBC connections and result sets from an SQLJ program may be necessary for finer control. The two paradigms interoperate seamlessly with each other.

It is possible to extract a JDBC `Connection` object from an SQLJ default connection context as follows:

```
DefaultContext cx1 =
    Oracle.getConnection("jdbc:oracle:oci8:@","raj","raj",true);
DefaultContext.setDefaultContext(cx1);
Connection conn = DefaultContext.getDefaultContext()
                                  .getConnection();
```

and for an SQLJ connection context to be initialized with a JDBC connection as shown here:

```
#sql context PortDB;

Connection conn = DriverManager.getConnection(
        "jdbc:oracle:oci8:book/book");
PortDB cx2 = new PortDB(conn);
```

The following is an SQLJ program that performs dynamic SQL using JDBC. A JDBC `Connection` object is extracted from the SQLJ `DefaultContext` object and a dynamic query is performed.

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import java.io.*;
import oracle.sqlj.runtime.*;

public class Dsqlj {
  public static void main (String args[])
     throws SQLException {

    DefaultContext cx1 =
        Oracle.getConnection("jdbc:oracle:oci8:@",
                             "book","book",true);
```

```
    DefaultContext.setDefaultContext(cx1);

    // Get a JDBC Connection object from an
    // SQLJ DefaultContext object
    Connection conn = DefaultContext.getDefaultContext()
                                    .getConnection();

    String sym = readEntry("Enter symbol substring: ")
                .toUpperCase();
    String query =
      "select cname,current_price,ask_price,bid_price "+
      "from   security " +
      "where symbol like '%" + sym + "%'";
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(query);

    while (rs.next())  {
      System.out.println("\n  Company Name  = " +
                         rs.getString(1));
      System.out.println("  Last sale at  = " + rs.getString(2));
      String ap = rs.getString(3);
      if (rs.wasNull())
        System.out.println("  Ask price     = null");
      else
        System.out.println("  Ask price     = " + ap);
      String bp = rs.getString(4);
      if (rs.wasNull())
        System.out.println("  Bid price     = null");
      else
        System.out.println("  Bid price     = " + bp);
    }
    rs.close();
    stmt.close();
  }
}
```

JDBC result sets and SQLJ iterators can also be easily transformed from one to the other. To convert an SQLJ iterator into a JDBC result set, use the `getResultSet` method on an iterator object, as follows:

```
QueryIterator q;
#sql q = {<sql query>};
ResultSet rs = q.getResultSet();
```

To convert the result set back into an SQLJ iterator, use the `CAST` operator, as follows:

```
ResultSet rs = ....;
#sql q = {CAST :rs};
```

6.8 Calling PL/SQL from Within SQLJ

When using SQLJ with Oracle, it is possible to embed PL/SQL anonymous blocks in the Java program. For example, the following program, containing an anonymous PL/SQL block, populates a table called `squares`, defined as follows:

```
create table squares (n number, nsquared number);
```

with the first 10 squares of natural numbers.

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import java.io.*;
import oracle.sqlj.runtime.*;

public class Anon {
  public static void main (String args[]) throws SQLException {
    DefaultContext cx1 =
        Oracle.getConnection("jdbc:oracle:oci8:@",
                             "book","book",true);
    DefaultContext.setDefaultContext(cx1);
    #sql {
      declare
        n number;
      begin
        delete from squares;
        n := 1;
        while (n <= 10) loop
          insert into squares values (n,n*n);
          n := n + 1;
        end loop;
```

```
      end;
    };
  }
}
```

It is also possible for SQLJ programs to make calls to PL/SQL stored procedures and functions. Consider the following PL/SQL stored procedure:

```
create or replace procedure latestTransactionDate
  (midd in member.mid%type, ldate out date) is
begin
  select max(trans_date)
  into   ldate
  from   transaction
  where  mid = midd;
end;
```

This stored procedure takes as input a member ID, `midd`, and returns the latest transaction date for that member in the output parameter `ldate`. This stored procedure can be called in an SQLJ program as follows:

```
String m = "10000";
java.sql.Date lastDate;
#sql {CALL latestTransactionDate(:in m, :out lastDate)};
```

Note the `IN` and `OUT` qualifiers for the parameters and also the data type matching between Java types and the corresponding PL/SQL types in the stored procedure. The whole SQLJ program that makes a call to the stored procedure is shown here:

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import java.io.*;
import oracle.sqlj.runtime.*;

public class Plsql1 {
  public static void main (String args[]) throws SQLException {
    DefaultContext cx1 =
        Oracle.getConnection("jdbc:oracle:oci8:@",
                             "book","book",true);
    DefaultContext.setDefaultContext(cx1);

    String m = readEntry("Member ID: ");
```

```
    java.sql.Date lastDate;
    #sql {CALL latestTransactionDate(:in m, :out lastDate)};
    System.out.println("The last transaction date is " +
                       lastDate);
  }
}
```

PL/SQL stored functions are invoked in a similar manner as stored procedures, except that the `CALL` token is replaced by the `VALUES` token and the function call is enclosed within parentheses. Also, the result of the function call is assigned to a Java variable. Consider the PL/SQL stored function called `avgPP`, which takes as input a member ID and a security symbol and computes the average purchase price that the member paid for the current shares held in his or her portfolio. For all the `buy` transactions the member has for a particular security, a weighted average price is computed. Finally, a commission of 1% is applied to the total amount to compute the average purchase price. The PL/SQL stored function is

```
create or replace function avgPP(
    mid in member.mid%type,
    sym in security.symbol%type)
  return transaction.price_per_share%type as
  cursor c1 is
    select trans_type, quantity, price_per_share
    from transaction
    where mid = avgPP.mid and
          symbol = sym
    order by trans_date;
  q transaction.quantity%type := 0.0;
  a transaction.price_per_share%type := 0.0;
begin
  for c1_rec in c1 loop
    if (c1_rec.trans_type = 'buy') then
      a := ((q * a) +
            (c1_rec.quantity*c1_rec.price_per_share))/
           (q + c1_rec.quantity);
      q := q + c1_rec.quantity;
    else
      q := q - c1_rec.quantity;
    end if;
  end loop;
```

```
  return (1.01 * a);  -- Commission of 1%
end;
```

A call to the `avgPP` function in SQLJ is

```
String m = readEntry("Member ID: ");
String sym = readEntry("Symbol: ");
double pp;
#sql pp = { VALUES(avgPP(:in m, :in sym)) };
```

Notice the `IN` tokens, the `VALUES` token, and the assignment of the result of the function call to a Java variable. An SQLJ program that illustrates the stored function call is shown here:

```
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.sql.*;
import java.io.*;
import oracle.sqlj.runtime.*;

public class Plsql2 {
  public static void main (String args[])
     throws SQLException {
    DefaultContext cx1 =
        Oracle.getConnection("jdbc:oracle:oci8:@",
                             "book","book",true);
    DefaultContext.setDefaultContext(cx1);

    String m = readEntry("Member ID: ");
    String sym = readEntry("Symbol: ");
    double pp;
    #sql pp = { VALUES(avgPP(:in m, :in sym)) };
    System.out.println("The average purchase price " +
        "member " + m + " paid for security " +
        sym + " is " + pp);
  }
}
```

6.9 Investment Portfolio Database Application

In this section, an application program that interfaces with the investment portfolio database is presented. A few of the methods of this application have already been discussed earlier in the chapter. The remaining are presented here.

The program begins by presenting the following main menu:

```
(a) Member Login
(b) New Member Sign-in
(q) Quit
```

A new member can use option `(b)` to create a new account. This option prompts the new member for name, address, email, and password. It creates a new account and informs the new member of the account number.

An existing member can use option `(a)` to log into his or her account. The member is prompted for an account number and password. Upon successful login, the member is shown the following menu of options:

```
(a) View Portfolio
(b) Print Monthly Report
(c) Update Your Record
(d) Price Quote
(e) Buy a Stock
(f) Sell a Stock
(q) Quit
```

The `main` *and* `printMenu` *Methods*

The `main` method, along with the `printMenu` method, are

```
public static void main (String args[])
  throws Exception,IOException,SQLException {

  String user = readEntry("userid  : ");
  String pass = readEntry("password: ");
  DefaultContext cx1 =
      Oracle.getConnection("jdbc:oracle:oci8:@",
                           user,pass,false);
  DefaultContext.setDefaultContext(cx1);

  boolean done = false;
  do {
```

```
      printMenu();
      String ch = readEntry("Type in your option: ");
      switch (ch.charAt(0)) {
        case 'a': memberLogIn();
                  break;
        case 'b': newMember();
                  break;
        case 'q': done = true;
                  break;
        default : System.out.println("Invalid option");
      }
    } while(!done);
  }

  private static void printMenu() {
    System.out.println("\nINVESTMENT PORTFOLIO " +
                       "TRACKING SYSTEM \n");
    System.out.println("(a) Member Log in ");
    System.out.println("(b) New Member Sign in ");
    System.out.println("(q) Quit. \n");
  }
```

After prompting the user for the Oracle user ID and password,[4] the `main` method opens a connection to the Oracle schema for the specified user and presents the main menu. Notice that the `autoCommit` mode is set to `false` in the `Oracle.getConnection` method call. It reads the user's option and calls the appropriate method.

The `memberLogin` *and* `printMenu1` *Methods*

The `memberLogIn` method prompts the user for an account number and password. It then verifies that the account number exists and that the password provided is correct. This is accomplished by a simple SQL query against the `member` table using the `select-into` statement. After the account number and password are verified, a member menu of options is presented. The program then reads the user's selected option and processes it appropriately by calling the corresponding method. The `memberLogIn` and `printMenu1` methods are

4. Note that there are two sets of user IDs and passwords. The first is the Oracle user ID and password, which enables the user to use the application program, and the second is the member's account ID and password, which enables the member to access his or her account.

```
private static void memberLogIn()
    throws SQLException, IOException {
  String mmid1,pass1;
  String mmid2 = readEntry("Account#: ");
  String pass2 = readEntry("Password: ");

  try {
    #sql {select mid, password
          into   :mmid1, :pass1
          from   member
          where  mid = :mmid2 and password = :pass2};
  } catch (SQLException e) {
      System.out.println("Invalid Account/Password");
      return;
    }

  boolean done = false;
  do {
    printMenu1();
    String ch = readEntry("Type in your option: ");
    switch (ch.charAt(0)) {
      case 'a': viewPortfolio(mmid1);
                break;
      case 'b': printReport(mmid1);
                break;
      case 'c': updateMember(mmid1);
                break;
      case 'd': getPriceQuote();
                break;
      case 'e': buyStock(mmid1);
                break;
      case 'f': sellStock(mmid1);
                break;
      case 'q': done = true;
                break;
      default : System.out.println("Invalid option ");
    }
  } while(!done);
}
```

```
private static void printMenu1() {
  System.out.println("\n    MEMBER OPTIONS \n");
  System.out.println("(a) View Portfolio ");
  System.out.println("(b) Print Monthly Report ");
  System.out.println("(c) Update Your Record ");
  System.out.println("(d) Price Quote ");
  System.out.println("(e) Buy a Stock ");
  System.out.println("(f) Sell a Stock ");
  System.out.println("(q) Quit \n");
}
```

The `newMember` *Method*

The `newMember` method prompts the user for all the pertinent information and then performs an SQL `insert` statement. The sequence object `m_seq` is used to generate a new account number. The code for `newMember` is

```
private static void newMember()
     throws SQLException, IOException {
  String pass, fn, ln, addr, email;
  double cash;

  fn    = readEntry("Enter first name: ");
  ln    = readEntry("Enter last name: ");
  addr  = readEntry("Enter address: ");
  email = readEntry("Enter email: ");
  pass  = readEntry("Enter password : ");
  cash  = Double.valueOf(readEntry("Enter initial cash : "))
            .doubleValue();
  try {
    String mmid;
    #sql {select m_seq.nextval into :mmid from dual};
    #sql {insert into member values
            (:mmid,:pass,:fn,:ln,:addr,:email,:cash)};
    #sql {commit};
    System.out.println("\nYour Account Number is " + mmid);
  } catch (SQLException e) {
      System.out.println("Could not add member");
      System.out.println("Message:"+e.getMessage());
      return;
    }
}
```

The `viewPortfolio` *Method*

The `viewPortfolio` method takes as input the member ID and prints the member's portfolio of stocks owned, their purchase prices, their current values, and the gain or loss. A summary for the entire portfolio is also produced. A sample portfolio view is shown in Figure 6.2.

Figure 6.2 Portfolio view output.

```
                        MY PORTFOLIO

Symbol  Shares   Current    Market    Purchase    Gain    %Gain
                  PPS       Value       Price
-----------------------------------------------------------------
ORCL    100.00    23.25     2325.00   2708.06   -383.06   -14.14
SEG     100.00    30.00     3000.00   3244.62   -244.62    -7.53
-----------------------------------------------------------------
  Security Value:           5325.00   5952.68   -627.68   -10.54
  Cash Balance:            94047.33
  Account Value:           99372.33
-----------------------------------------------------------------
```

The query used to compute the current portfolio for a member is as follows:

```
select s.symbol, p.quantity, s.current_price,
       (p.quantity * s.current_price) MarketValue,
       (p.quantity * avgPP(:mid,p.symbol)) PurchasePrice
from   security s, portfolio p
where  p.mid = :mid and s.symbol = p.symbol;
```

This query uses the SQL view `portfolio`, defined in Section 2.5. It makes use of a PL/SQL stored function called `avgPP`, which was introduced in Section 6.8. The `viewPortfolio` method uses a named SQLJ iterator called `PView`, shown here:

```
#sql iterator PView(String symbol, double quantity,
   double current_price, double MarketValue,
   double PurchasePrice);
```

This iterator is used to retrieve all the stocks currently owned by the member, and the information retrieved is formatted and sent to the screen. The `viewPortfolio` method is shown next:

```
private static void viewPortfolio(String mid)
     throws SQLException, IOException {
  String mmid;
  try {
    #sql { select distinct mid
           into   :mmid
           from   portfolio
           where  mid = :mid };
  } catch (SQLException e) {
      System.out.println("Empty Portfolio");
      return;
    }

  PView p = null;
  #sql p =
    {select s.symbol,p.quantity,s.current_price,
            (p.quantity*s.current_price) MarketValue,
            (p.quantity*avgPP(:mid,p.symbol))
            PurchasePrice
     from   security s, portfolio p
     where  p.mid=:mid and s.symbol=p.symbol};

  double cash;
  #sql {select cash_balance
        into   :cash
        from   member
        where  mid = :mid};

  // Print Report Header
  writeSpaces(30);
  System.out.println("MY PORTFOLIO");
  System.out.print("Symbol"); writeSpaces(5);
  System.out.print("Shares"); writeSpaces(3);
  System.out.print("Current"); writeSpaces(4);
  System.out.print("Market"); writeSpaces(6);
  System.out.print("Purchase"); writeSpaces(6);
  System.out.print("Gain"); writeSpaces(7);
  System.out.println("%Gain"); writeSpaces(21);
  System.out.print("PPS"); writeSpaces(7);
  System.out.print("Value"); writeSpaces(8);
```

```
System.out.println("Price");
writeDashes(76); System.out.println();

double gainLoss, pGL;
double total_pp = 0.0, total_mv = 0.0;

while(p.next()) {
  writeEntryLeft(p.symbol(),8);
  writeEntryRight(twoDigit(p.quantity()),9);
  writeEntryRight(twoDigit(p.current_price()),9);
  writeEntryRight(twoDigit(p.MarketValue()),12);
  total_mv += p.MarketValue();
  writeEntryRight(twoDigit(p.PurchasePrice()),12);
  total_pp += p.PurchasePrice();
  gainLoss = p.MarketValue() - p.PurchasePrice();
  pGL = (gainLoss/p.PurchasePrice() ) * 100;
  writeEntryRight(twoDigit(gainLoss),12);
  writeEntryRight(twoDigit(pGL),12);
  System.out.println();
}

p.close();

writeDashes(76); System.out.println();
System.out.print("  Security Value: ");
writeSpaces(8);
writeEntryRight(twoDigit(total_mv),12);
writeEntryRight(twoDigit(total_pp),12);
writeEntryRight(twoDigit(total_mv - total_pp),12);
writeEntryRight(twoDigit(((total_mv - total_pp)/
                          total_pp)*100),12);
System.out.println();

System.out.print("  Cash Balance: ");
writeSpaces(12);
writeEntryRight(twoDigit(cash),10);
System.out.println();

System.out.print("  Account Value: ");
```

```
    writeSpaces(11);
    writeEntryRight(twoDigit(cash + total_mv),10);
    System.out.println();

    writeDashes(76);
  }
```

The `updateMember` *Method*

The `updateMember` method allows the member to make changes to his or her password, address, or email. The method first performs a query to obtain the current password, address, and email of the member. It then prompts the user for new information and issues an SQL `update` statement to make the changes. The code follows:

```
  private static void updateMember(String mmid)
       throws SQLException, IOException {

    String password, address, email, answer;
    try {
      #sql { select password, address, email
             into   :password,:address,:email
             from   member
             where  mid = :mmid };
    } catch (SQLException e) {
        System.out.println("Invalid member account");
        return;
      }

    boolean change = false;
    System.out.println("password : " + password);
    answer = readEntry("Change password?(y/n):").toLowerCase();
    if (answer.equals("y")) {
      password = readEntry("Enter new password : ");
      change = true;
    }

    System.out.println("address : " + address);
    answer = readEntry("Change address?(y/n):").toLowerCase();
```

```
    if (answer.equals("y")) {
      address = readEntry("New address : ");
      change = true;
    }

    System.out.println("email : " + email);
    answer = readEntry("Change email?(y/n):").toLowerCase();
    if (answer.equals("y")) {
      email = readEntry("New email : ");
      change = true;
    }

    if (change) {
      #sql { update member
             set password = :password,
                 address = :address,
                 email = :email
             where mid = :mmid };
      #sql {commit};
      System.out.println("Updated successfully ");
    }
    else
      System.out.println("No changes made");
  }
```

The `getPriceQuote` *and* `getPriceQuoteBySymbol` *Methods*

The `getPriceQuote` method allows the member to obtain a quote on a security based on the symbol or the company name of the security. It prompts the user for the option and calls the appropriate method for obtaining the quote. The `getPriceQuoteByCname` method, for obtaining a price quote for a security given a substring of the company name, was presented earlier in this chapter. The `getPriceQuoteBySymbol` method prompts the member for the exact security symbol and then performs a simple SQL query using the `select-into` statement. The code for these two methods follows:

```
  private static void getPriceQuote()
         throws SQLException, IOException {

    System.out.println("(a) Look up by symbol");
    System.out.println("(b) Look up by company name");
```

```
  String ch =
    readEntry("Type in your option: ").toLowerCase();

  String sym="",cn="";
  switch (ch.charAt(0)) {
    case 'a' :
      sym = readEntry("Enter symbol : ").toUpperCase();
      getPriceQuoteBySymbol(sym);
      break;
    case 'b' :
      cn = "%" + readEntry("Enter search string: ")
                 .toUpperCase() + "%";
      getPriceQuoteByCname(cn);
      break;
  }
}

public static void getPriceQuoteBySymbol(String sym)
       throws SQLException, IOException {
  double cp=0.0;
  Double ap=null,bp=null;
  String cn = "";

  try {
    #sql { select cname,current_price,ask_price,bid_price
           into   :cn,:cp,:ap,:bp
           from   security
           where  symbol = :sym };
  } catch (SQLException e) {
      System.out.println("Invalid symbol.");
      return;
    }
  System.out.println("\n  Company Name  = " + cn);
  System.out.println("  Last sale at  = " + cp);
  if (ap == null)
    System.out.println("  Ask price     = null");
  else
    System.out.println("  Ask price     = " + ap);
  if (bp == null)
    System.out.println("  Bid price     = null");
```

```
    else
      System.out.println("  Bid price     = " + bp);
  }
```

The `buyStock` *and* `buyConfirmation` *Methods*

The `buyStock` method first prompts the member for the security symbol. It then obtains the current bidding price from the database. If the bidding price is `null`, it uses the current selling price. After this, the method obtains the number of shares from the user. Using this information, it computes the total cost of this transaction, including a 1% commission on the total value. The current cash balance for the member is obtained to see if the member has enough funds to pay for the transaction. The method then asks the member for a confirmation by calling the `buyConfirmation` method. Upon confirmation, a new row is added to the `transaction` table for this transaction. The cash balance in the `member` table is also updated. The code for `buyStock` and `buyConfirmation` follows:

```
  public static void buyStock(String mid)
      throws SQLException, IOException {

    double currentprice = 0.0;
    Double bidprice;

    String symbol  = readEntry("Symbol  : ");
    try {
      #sql { select current_price, bid_price
             into   :currentprice, :bidprice
             from   security
             where  symbol = :symbol };
    } catch (SQLException e) {
        System.out.println("Stock information not found");
        return;
      }

    double quantity =
      Double.valueOf(readEntry("Quantity: ")).doubleValue();

    double price,total, cash;
    if (bidprice == null)
      price = currentprice;
    else
```

```
    price = bidprice.doubleValue();

  double commission =
    Double.valueOf(twoDigit(0.01 * (price * quantity)))
          .doubleValue();
  total =
    Double.valueOf(twoDigit((price * quantity) + commission))
          .doubleValue();

  #sql { select cash_balance
         into   :cash
         from   member
         where  mid = :mid };

  if (total > cash) {
    System.out.println("Sorry, not enough money!");
    return;
  }

  if(!buyConfirmation(symbol,quantity,price)){
    System.out.println("Transaction was not processed");
    return;
  }

  #sql { insert into transaction values
           (:mid,:symbol,sysdate,'buy',:quantity,
            :price,:commission,:total) };
  #sql {commit};

  #sql { update member
         set    cash_balance = cash_balance - :total
         where  mid = :mid };
  #sql {commit};

  System.out.println("Successful Buy Transaction");
}

public static boolean buyConfirmation(
         String symbol, double quantity,
         double price_per_share) {
```

```
    double total = price_per_share * quantity;
    System.out.println("\n\t\tConfirmation:");
    System.out.println(quantity + " shares of " + symbol +
        " at $" + price_per_share + " per share.");
    System.out.println("\n");
    System.out.println("\tTotal price of shares:    "  +
                       total+"\n");
    System.out.println("\tCommission:              +"
                        + (0.01 * total));
    System.out.println("\t                         " +
                       " -------------");
    System.out.println("\tTotal:                    $" +
                        (1.01 * total));
    System.out.println();
    for (;;) {
      String answer =
        readEntry("Accept?  (Y or N): ").toUpperCase();
      switch (answer.charAt(0)) {
        case 'Y': return true;
        case 'N': return false;
        default:   break;
      }
    }
  }
```

The `sellStock` and `sellConfirmation` methods are very similar.

Exercises

Investment Portfolio Database Problems

6.1 Write an SQLJ program that populates the `security` table by fetching data from the following Web address:

```
http://www.quicken.com/investments/quotes/?symbol=XXXX
```

where `XXXX` is the security symbol. This Web page contains information about the company in detail; however, only the name of the company and the current price need to be retrieved. The `ask_price` and `bid_price` values should be set to `null`. To

solve this problem, you should import the `java.net` package and use the following Java method that fetches the HTML content of a Web page into a Java string variable:

```
private static String getHTMLContent(URL hp) throws Exception {
  BufferedReader data = new BufferedReader(new
    InputStreamReader(hp.openStream()));
  String line, sHtml="";

  while ((line = data.readLine()) != null)
    sHtml += line + "\n";
  data.close();
  return sHtml;
}
```

This method should be invoked as follows:

```
URL hp = new URL(
  "http://www.quicken.com/investments/quotes/?symbol="+XXXX);
String sHtml = getHTMLContent(hp);
```

where `XXXX` is a string variable containing the security symbol. To retrieve data for more than one security, you should create a constant array of security symbols within the Java program, loop through the array and, for each symbol in the array, fetch data from the Web page and store it in the database.

6.2 Write an SQLJ program that updates the `security` table for the current, ask, and bid prices by fetching data from the following Web address:

```
http://www.quicken.com/investments/quotes/?symbol=XXXX
```

where `XXXX` is the security symbol. This Web page contains a variety of information about the price quotes for the company. However, only the current, ask, and bid prices need to be retrieved.

6.3 Write an SQLJ program to implement the following menu for analysts:

```
(1) Rate a stock
(2) Update a stock rating
(3) Quit
```

The program should begin by prompting the analyst for an ID and password. After verifying the ID and password, the program should display the menu. To rate a stock, the program should prompt the analyst for the symbol and the rating. To update a stock rating, the program should prompt the analyst for the symbol, display the current rating by the analyst, and prompt the analyst for a new rating.

6.4 Implement the following two additional menu options for the `MEMBER OPTIONS` of the application presented in Section 6.9:

```
(g) View Ratings for a Security
(h) View Five Top-Rated Securities
```

The program should prompt the user for the security symbol for the view rating option and then display all the ratings by analysts for that symbol. The output should resemble the following:

```
Symbol: ORCL
Company: Oracle Corporation
Ratings:  Strong Buy  (rating = 1) : *****
          Buy         (rating = 2) : **
          Hold        (rating = 3) : **
          Sell        (rating = 4) :
          Strong Sell (rating = 5) :
          Consensus:       1.67
```

The number of stars after each rating is the number of analysts rating the security with that particular rating. The consensus mean is the weighted mean of the ratings. The View Five Top-Rated Securities option should display the top five securities based on the consensus mean value, in increasing order of consensus mean.