JDBC^{TM} RowSet Implementations Tutorial

Maydene Fisher with contributions from
Jonathan Bruce, Amit Handa & Shreyas Kaushik

Sun Microsystems Inc.
4150 Network Circle
Santa Clara, CA 95054
USA

Revision 1.0

Send comments to jdbc@sun.com

1

RowSet Overview

A JDBC RowSet object holds tabular data in a way that makes it more flexible and easier to use than a result set. Sun Microsystems has defined five RowSet interfaces for some of the more popular uses of a RowSet object, and the Java Community Process has produced standard reference implementations for these five RowSet interfaces. In this tutorial you will learn how easy it is to use these reference implementations, which together with the interfaces are part of the Java™ 2 Platform, Standard Edition 5.0 (J2SE™ 5.0).

Sun provides the five versions of the RowSet interface and their implementations as a convenience for developers. Developers are free write their own versions of the javax.sql.RowSet interface, to extend the implementations of the five RowSet interfaces, or to write their own implementations. However, many programmers will probably find that the standard reference implementations already fit their needs and will use them as is.

This chapter gives you an overview of the five RowSet interfaces, and the succeeding chapters walk you through how to use each of the reference implementations.

What Can RowSet Objects Do?

All RowSet objects are derived from the ResultSet interface and therefore share its capabilities. What makes JDBC RowSet objects special is that they add new capabilities, which you will learn to use in the following chapters.

Function as a JavaBeans™ Component

All RowSet objects are JavaBeans™ components. This means that they have the following:

- Properties
- The JavaBeans notification mechanism

Properties

All RowSet objects have properties. A property is a field that has the appropriate getter and setter methods in the interface implementation. For example, the Base-RowSet abstract class, a convenience class in the JDBC RowSet Implementations, provides the methods for setting and getting properties, among other things. All of the RowSet reference implementations extend this class and thereby have access to these methods. If you wanted to add a new property, you could add the getter and setter methods for it to your implementation. However, the BaseRowSet class provides more than enough properties for most needs.

Just because there are getter and setter methods for a property does not mean that you must set a value for every property. Many properties have default values, and setting values for others is optional if that property is not used. For example, all RowSet objects must be able to obtain a connection to a data source, which is generally a database. Therefore, they must have set the properties needed to do that. You can get a connection in two different ways, using the DriverManager mechanism or using a DataSource object. Both require the username and password properties to be set, but using the DriverManager requires that the url property be set, whereas using a DataSource object requires that the dataSourceName property be set.

The default value for the type property is ResultSet.TYPE_SCROLL_INSENSITIVE and for the concurrency property is ResultSet.CONCUR_UPDATABLE. If you are working with a driver or database that does not offer scrollable and updatable ResultSet objects, you can use a RowSet object populated with the same data as a ResultSet object and thereby effectively make that ResultSet object scrollable and updatable. You will see how this works in the chapter “JdbcRowSet.”

The following BaseRowSet methods set other properties:

- setCommand
- setEscapeProcessing—default is on
- setFetchDirection
- setFetchSize

- setMaxFieldSize
- setMaxRows
- setQueryTimeout—default is no time limit
- setShowDeleted—default is not to show deleted rows
- setTransactionIsolation—default is not to see dirty reads
- setTypeMap—default is null

You will see a lot more of the command property in future chapters.

Event Notification

RowSet objects use the JavaBeans event model, in which registered components are notified when certain events occur. For all RowSet objects, three events trigger notifications:

1. A cursor movement
2. The update, insertion, or deletion of a row
3. A change to the entire RowSet contents

The notification of an event goes to all *listeners*, components that have implemented the RowSetListener interface and have had themselves added to the RowSet object's list of components to be notified when any of the three events occurs.

A listener could be a GUI component such as bar graph. If the bar graph is tracking data in a RowSet object, it would want to know the new data values whenever the data changed. It would therefore implement the RowSetListener methods to define what it will do when a particular event occurs. Then it also needs to be added to the RowSet object's list of listeners. The following line of code registers the bar graph component *bg* with the RowSet object *rs*.

```
rs.addListener(bg);
```

Now *bg* will be notified each time the cursor moves, a row is changed, or all of *rs* gets new data.

Add Scrollability or Updatability

Some DBMSs do not support result sets that are scrollable, and some do not support result sets that are updatable. If a driver for that DBMS does not add scrollability or updatability, you can use a RowSet object to do it. A RowSet object is scrollable and updatable by default, so by populating a RowSet object with the contents of a result set, you can effectively make the result set scrollable and updatable.

Kinds of RowSet Objects

A RowSet object is considered either connected or disconnected. A *connected* RowSet object uses a driver based on JDBC technology ("JDBC driver") to make a connection to a relational database and maintains that connection throughout its life span. A *disconnected* RowSet object makes a connection to a data source only to read in data from a ResultSet object or to write data back to the data source. After reading data from or writing data to its data source, the RowSet object disconnects from it, thus becoming "disconnected." During much of its life span, a disconnected RowSet object has no connection to its data source and operates independently. The next two sections tell you what being connected or disconnected means in terms of what a RowSet object can do.

Connected RowSet Objects

Only one of the standard RowSet implementations is a connected RowSet: JdbcRowSet. Being always connected to a database, it is most similar to a ResultSet object and is often used as a wrapper to make an otherwise nonscrollable and readonly ResultSet object scrollable and updatable.

As a JavaBeans component, a JdbcRowSet object can be used, for example, in a GUI tool to select a JDBC driver. A JdbcRowSet object can be used this way because it is effectively a wrapper for the driver that obtained its connection to the database.

Disconnected RowSet Objects

The other four implementations are disconnected RowSet implementations. Disconnected RowSet objects have all the capabilities of connected RowSet objects plus they have the additional capabilities available only to disconnected RowSet objects. For example, not having to maintain a connection to a data source makes disconnected RowSet objects far more lightweight than a JdbcRowSet object or a ResultSet object. Disconnected RowSet objects are also serializable, and the combination of being both serializable and lightweight makes them ideal for sending data over a network. They can even be used for sending data to thin clients such as PDAs and mobile phones.

The CachedRowSet interface defines the basic capabilities available to all disconnected RowSet objects. The other three are extensions of it providing more specialized capabilities. The following outline shows how they are related.

- CachedRowSet
  - WebRowSet
    - JoinRowSet
    - FilteredRowSet

A CachedRowSet object has all the capabilities of a JdbcRowSet object plus it can also do the following:

- Obtain a connection to a data source and execute a query
- Read the data from the resulting ResultSet object and populate itself with that data
- Manipulate data and make changes to data while it is disconnected
- Reconnect to the data source to write changes back to it
- Check for conflicts with the data source and resolve those conflicts

A WebRowSet object has all the capabilities of a CachedRowSet object plus it can also do the following:

- Write itself as an XML document
- Read an XML document that describes a WebRowSet object

A JoinRowSet object has all the capabilities of a WebRowSet object (and therefore also a CachedRowSet object) plus it can also do the following:

- Form the equivalent of an SQL JOIN without having to connect to a data source

A FilteredRowSet object likewise has all the capabilities of a WebRowSet object (and therefore also a CachedRowSet object) plus it can also do the following:

- Apply filtering criteria so that only selected data is visible. This is equivalent to executing a query on a RowSet object without having to use a query language or connect to a data source.

The following chapters walk you through how to use the reference implementations for each of the interfaces introduced in this chapter.

2

JdbcRowSet

A JdbcRowSet object is basically an enhanced ResultSet object. It maintains a connection to its data source, just as a ResultSet object does. The big difference is that it has a set of properties and a listener notification mechanism that make it a JavaBeans™ component. This chapter covers properties, and the chapter “CachedRowSet” covers the listener notification mechanism in the section "Notifying Listeners," on page 29.

One of the main uses of a JdbcRowSet object is to make a ResultSet object scrollable and updatable when it does not otherwise have those capabilities.

In this chapter, you will learn how to:

- Create a JdbcRowSet object
- Set properties
- Move the cursor to different rows
- Update data
- Insert a new row
- Delete a row

Creating a JdbcRowSet Object

You can create a JdbcRowSet object in two ways:

- By using the reference implementation constructor that takes a ResultSet object

- By using the reference implementation default constructor

Passing a ResultSet Object

The simplest way to create a JdbcRowSet object is to produce a ResultSet object and pass it to the JdbcRowSetImpl constructor. Doing this not only creates a JdbcRowSet object but also populates it with the data in the ResultSet object.

As an example, the following code fragment uses the Connection object *con* to create a Statement object, which then executes a query. The query produces the ResultSet object *rs,* which is passed to the constructor to create a new JdbcRowSet object initialized with the data in *rs*.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(select * from COFFEES);
JdbcRowSet jdbcRs = new JdbcRowSetImpl(rs);
```

Note that because no arguments are passed to the method createStatement, any ResultSet objects it produces will be neither scrollable nor updatable. As a result, you can move the cursor for *rs* only forward, and you cannot make changes to the data in *rs*. However, we now have the data from *rs* in *jdbcRs*, and you can move the cursor for *jdbcRs* to any position and can also modify the data in *jdbcRs*.

Because the newly created JdbcRowSet object *jdbcRs* contains exactly the same data as *rs*, it can be considered a wrapper for *rs*. Assume that Table 2–1COFFEES represents the data in both *rs* and *jdbcRs*.

Table 2–1 COFFEES

COF_ID	COF_NAME	SUP_ID	PRICE
1250	Colombian	101	7.99
1300	French_Roast	49	8.99
1800	Espresso	150	9.99
2250	Colombian_Decaf	101	8.99

The column names mean the following:

COF_ID = Coffee Identification Number—INTEGER
COF_NAME = Coffee Name—VARCHAR(20)
SUP_ID = Supplier Identification Number—INTEGER
PRICE = Price per pound of coffee—DECIMAL(6,2)

Being scrollable and updatable are only two of the default properties of a escape Processing object. In addition to populating *jdbcRs* with the data from *rs*, the constructor also sets the following properties with the following values:

- type—ResultSet.TYPE_SCROLL_INSENSITIVE (has a scrollable cursor)
- concurrency—ResultSet.CONCUR_UPDATABLE (can be updated)
- escapeProcessing—true (the driver will do escape processing)
- maxRows—0 (no limit on the number of rows)
- maxFieldSize—0 (no limit on the number of bytes for a column value; applies only to columns that store BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR values)
- queryTimeout—0 (has no time limit for how long it takes to execute a query)
- showDeleted—false (deleted rows are not visible)
- transactionIsolation—Connection.TRANSACTION_READ_COMMITTED (reads only data that has been committed)
- typeMap—null (the type map associated with a Connection object used by this RowSet object is null)

The main thing you need to remember from this list is that a JdbcRowSet and all other RowSet objects are scrollable and updatable unless you set different values for those properties.

Using the Default Constructor

The following line of code creates an empty JdbcRowSet object.

```
JdbcRowSet jdbcRs2 = new JdbcRowSetImpl();
```

All of the reference implementation constructors assign the default values for the porperties listed in the section “Passing a ResultSet Object,” so although *jdbcRs2* has no data yet, it has the same properties set with default values as *jdbcRs*. To populate *jdbcRs2* with data, you need a ResultSet object with the desired data. This means you need to get a connection and execute a query, which requires your setting the properties needed for getting a connection and setting the query to be executed. You will see how to set these properties in the next section.

Setting Properties

The section “Passing a ResultSet Object” lists the properties that are set by default when a new RowSet object is created. If you use the default constructor, you need to set some additional properties before you can populate your new JdbcRowSet object with data.

In order to get its data, a JdbcRowSet object first needs to connect to a database. The following four properties hold information used in obtaining a connection to a database.

- username—the name a user supplies to a database as part of gaining access
- password—the user’s database password
- url—the JDBC URL for the database to which the user wants to connect
- datasourceName—the name used to retrieve a DataSource object that has been registered with a JNDI naming service

As was mentioned in the chapter “Overview,” which of these properties you need to set depends on how you are going to make a connection. The preferred way is to use a DataSource object, but it may not be practical for some readers to register a DataSource object with a JNDI naming service, which is generally done by a person acting in the capacity of a system administrator. Therefore, the code examples all use the DriverManager mechanism to obtain a connection, for which you use the url property and not the datasourceName property.

The following lines of code set the username, password, and url properties so that a connection can be obtained using the DriverManager mechanism. (You will find the JDBC URL to set as the value for the url property in the documentation for your JDBC driver.)

```
jdbcRs.setUsername("hardy");
jdbcRs.setPassword("oursecret");
jdbcRs.setUrl("jdbc:mySubprotocol:mySubname");
```

Another property that you must set is the command property. This property is the query that determines what data the JdbcRowSet object will hold. For example, the following line of code sets the command property with a query that produces a ResultSet object containing all the data in the table COFFEES.

```
jdbcRs.setCommand("select * from COFFEES");
```

Once you have set the command property and the properties necessary for making a connection, you are ready to populate *jdbcRs* with data. You can do this by simply calling the execute method.

```
jdbcRs.execute();
```

The execute method does many things for you behind the scenes.

1. It makes a connection to the database using the values you assigned to the url, username, and password properties.
2. It executes the query you set for the command property.
3. It reads the data from the resulting ResultSet object into *jdbcRs*.

At this point, *jdbcRs* and *jdbcRs2* should be identical.

Setting Parameters for the Command

In the preceding code fragments, we used a command that selected all of the data in the table COFFEES. If you wanted a JdbcRowSet object populated with only some of the data, you would need to use a where clause. For example, the query in the following line of code selects the coffee name and price for coffees whose price is greater than 7.99.

```
select COF_NAME, PRICE from COFFEES where PRICE > 7.99;
```

For more flexibility, you could use a placeholder parameter instead of 7.99. A placeholder parameter is a question mark (“?”) used in place of a literal value.

```
select COF_NAME, PRICE from COFFEES where PRICE > ?;
```

In this case, you have to supply the value for the placeholder parameter before you can execute the query. A query with placeholder parameters is a PreparedStatement object, and you use the equivalent of PreparedStatement setter methods to supply a placeholder parameter value, as is done in the following line of code. The first argument is the ordinal position of the placeholder parameter, and the second argument is the value to assign to it. When there is only one placeholder parameter, its ordinal position is, of course, one.

```
jdbcRs.setBigDecimal(1, new BigDecimal("8.99"));
```

If your query has two placeholder parameters, you must set values for both of them.

```
select COF_NAME, PRICE from COFFEES where PRICE > ? and SUP_ID = ?;

jdbcRs.setBigDecimal(1, new BigDecimal("8.99"));
jdbcRs.setInt(2, 101);
```

Note that ordinal position is the placeholder parameter's position in the command and has nothing to do with its column index in the ResultSet object or in *jdbcRs*.

Using a JdbcRowSet Object

You update, insert, and delete a row in a JdbcRowSet object the same way you update, insert, and delete a row in an updatable ResultSet object. Similarly, you navigate a JdbcRowSet object the same way you navigate a scrollable ResultSet object.

The Coffee Break chain of coffee houses acquired another chain of coffee houses and now has a legacy database that does not support scrolling or updating of a result set. In other words, any ResultSet object produced by this legacy database does not have a scrollable cursor, and the data in it cannot be modified. However, by creating a JdbcRowSet object populated with the data from a ResultSet object, you can, in effect, make the ResultSet object scrollable and updatable.

As mentioned previously, a JdbcRowSet object is by default scrollable and updatable. Because its contents are identical to those in a ResultSet object, operating on the JdbcRowSet object is equivalent to operating on the ResultSet object itself. And because a JdbcRowSet object has an ongoing connection to the database, changes it makes to its own data are also made to the data in the database.

Navigating a JdbcRowSet Object

A ResultSet object that is not scrollable can use only the next method to move its cursor forward, and it can only move the cursor forward from the first row to the last row. A JdbcRowSet object, however, can use all of the cursor movement methods defined in the ResultSet interface.

First, let's look at how the method next works.The Coffee Break owner wants a list of the coffees sold in his coffee houses and the current price for each. The following code fragment goes to each row in COFFEES and prints out the values in the columns COF_NAME and PRICE. The method next initially puts the cursor above the first row so that when it is first called, the cursor moves to the first row. On subsequent calls, this method moves the cursor to the next row. Because next returns true when there is another row and false when there are no more rows, it can be put into a while loop. This moves the cursor through all of the rows, repeatedly calling the method next until there are no more rows. As noted earlier, this is the only cursor movement method that a nonscrollable ResultSet object can call.

```
while (jdbcRs.next()) {
    String name = jdbcRs.getString("COF_NAME");
    BigDecimal price = jdbcRs.getBigDecimal("PRICE");
    System.out.println(name + "      " + price);
}
```

A JdbcRowSet object can call the method next , as seen in the preceding code fragment, and it can also call any of the other ResultSet cursor movement methods. For example, the following lines of code move the cursor to the fourth row in *jdbcRs* and then to the third row.

```
jdbcRs.absolute(4);
jdbcRs.previous();
```

The method previous is analogous to the method next in that it can be used in a while loop to traverse all of the rows in order. The difference is that you must move the cursor to after the last row, and previous moves the cursor toward the beginning.

```
jdbcRs.afterLast();
while (jdbcRs.previous()) {
    String name = jdbcRs.getString("COF_NAME");
    BigDecimal price = jdbcRs.getBigDecimal("PRICE");
    System.out.println(name + "      " + price);
}
```

The output for this piece of code will have the same data as the code fragment using the method next, except the rows will be printed in the opposite order, going from the last row to the first.

You will see the use of more cursor movement methods in the section on updating data.

Updating a Column Value

You update data in a JdbcRowSet object the same you update data in a ResultSet object.

Let's assume that the Coffee Break owner wants to raise the price for a pound of Espresso coffee. If he knows that Espresso is in the third row of *jdbcRs*, the code for doing this might look like the following:

```
jdbcRs.absolute(3);
jdbcRs.updateBigDecimal("PRICE", new BigDecimal("10.99"));
jdbcRs.updateRow();
```

The code moves the cursor to the third row, changes the value for the column "PRICE" to 10.99, and then updates the database with the new price. There are two things to note. First, for the first argument to the method updatetBigDecimal, we could have given the column number (which in this case is 4) instead of the column name.

Second, the data type for this column is an SQL DECIMAL, which is commonly used for columns with money values. The DECIMAL type takes two parameters, so the full data type for the column PRICE is DECIMAL(6, 2). The first parameter indicates the precision, or total number of digits. The second parameter indicates the number of digits to the right of the decimal point. So values in the PRICE column can have six digits, four digits before the decimal point and two digits after the decimal point. The recommended ResultSet getter method for retrieving values of type DECIMAL is getBigDecimal. Because BigDecimal is an Object type, you have to pass a BigDecimal object to the methods setBigDecimal and updateBigDecimal. This explains why the value being set is new BigDecimal("10.99"), which creates a BigDecimal object with the value 10.99. You can use a number as the parameter to new BigDecimal, but we use a String object because it is safer.

Databases vary in the names they use for data types, so if your database does not use DECIMAL, you can call the DatabaseMetaData method getTypeInfo to see what your database uses. The method getTypeInfo returns a ResultSet object with one row for each data type. The first column, TYPE_NAME, gives the name the database uses for a type. The second column, DATA_TYPE, gives the type code for the corresponding JDBC type (from the class java.sql.Types). The type code for DECIMAL is 3, so you want to use the name in the TYPE_NAME column of the row where the DATA_TYPE column value is 3. This is the type name to use in the CREATE TABLE statement for the data type of the column PRICE.

The third thing to note is that calling the method updateRow updates the database, which is true because *jdbcRs* has maintained its connection to the database. For disconnected RowSet objects, the situation is different, as you will see in the chapter “CachedRowSet.”

Inserting a Row

If the owner of the Coffee Break chain wants to add one or more coffees to what he offers, he will need to add one row to the COFFEES table for each new coffee, as is done in the following code fragment. You will notice that because *jdbcRs* is always connected to the database, inserting a row into a JdbcRowSet object is the same as inserting a row into a ResultSet object: You move to the insert row, use the appropriate updater method to set a value for each column, and call the method insertRow.

```
jdbcRs.moveToInsertRow();
jdbcRs.updateString("COF_NAME", "House_Blend");
jdbcRs.updateInt("SUP_ID", 49);
jdbcRs.updateBigDecimal("PRICE", new BigDecimal("7.99"));
jdbcRs.updateInt("SALES", 0);
jdbcRs.updateInt("TOTAL", 0);
jdbcRs.insertRow();
jdbcRs.moveToCurrentRow();

jdbcRs.moveToInsertRow();
jdbcRs.updateString("COF_NAME", "House_Blend_Decaf");
jdbcRs.updateInt("SUP_ID", 49);
jdbcRs.updateBigDecimal("PRICE", new BigDecimal("8.99"));
jdbcRs.updateInt("SALES", 0);
jdbcRs.updateInt("TOTAL", 0);
jdbcRs.insertRow();
jdbcRs.moveToCurrentRow();
```

When you call the method insertRow, the new row is inserted into *jdbcRs* and is also inserted into the database. The preceding code fragment goes through this process twice, so two new rows are inserted into *jdbcRs* and the database.

Deleting a Row

As is true with updating data and inserting a new row, deleting a row is just the same for a JdbcRowSet object as for a ResultSet object. The owner wants to discontinue selling French Roast decaf coffee, which is the last row in *jdbcRs*. In the following lines of code, the first line moves the cursor to the last row, and the second line deletes the last row from *jdbcRs* and from the database.

```
jdbcRs.last();
jdbcRs.deleteRow();
```

Code Sample

The following code sample, which you will find in the samples directory, is a complete, runnable program incorporating code fragments shown in this chapter. The code does the following:

1. Declares variables
2. Loads the driver and gets a connection
3. Creates the table COFFEES
4. Creates a Statement object and executes a query
5. Creates a new JdbcRowSet object initialized with the ResultSet object that was produced by the execution of the query
6. Moves to the third row and updates the PRICE column in that row
7. Inserts two new rows, one for HOUSE_BLEND and one for HOUSE_BLEND_DECAF
8. Moves to the last row and deletes it

Note that for some databases, you must create a BigDecimal object and then use its variable in an INSERT INTO statement to insert data in a database table. Further, the variable must be in a special format. For example, if the variable is b, it must be expressed as "+b+" in order to run successfully. You will see an example of this in the following sample code.

This code sample demonstrates moving the cursor to different rows and making changes to data.

==

```
import java.sql.*;
import javax.sql.rowset.*;
import com.sun.rowset.*;
import java.math.BigDecimal;

public class JdbcRowSetSample {
    public static void main(String args[]) {
```

```
String strUrl = "jdbc:datadirect:oracle://" +
            "129.158.229.21:1521;SID=ORCL9";
String strUserId = "scott";
tring strPassword = "tiger";
String className = "com.ddtek.jdbc.oracle.OracleDriver";

JdbcRowSet jdbcRs;
ResultSet rs;
Statement stmt;
Connection con;
BigDecimal b;

try {
   Class.forName(className);
} catch(java.lang.ClassNotFoundException e) {
   System.err.print("ClassNotFoundException: ");
   System.err.println(e.getMessage());
}

try {
   con = DriverManager.getConnection(
                  strUrl, strUserId, strPassword);
   con.setAutoCommit(false);

   stmt = con.createStatement(
                     ResultSet.TYPE_SCROLL_INSENSITIVE,
                     ResultSet.CONCUR_UPDATABLE);

   stmt.addBatch("drop table COFFEES");

   stmt.addBatch("create table COFFEES(COF_ID INTEGER, " +
         "COF_NAME VARCHAR(20), SUP_ID INTEGER, " +
         "PRICE DECIMAL(6,2))");

   b = new BigDecimal("7.99");
   stmt.addBatch("insert into COFFEES values " +
         "(1250, 'Colombian', 101, "+b+")");

    b = new BigDecimal("8.99");
   stmt.addBatch("insert into COFFEES values " +
         "(1300, 'French_Roast', 49, "+b+")");

    b = new BigDecimal("9.99");
   stmt.addBatch("insert into COFFEES values " +
         "(1800, 'Espresso', 150, "+b+")");

    b = new BigDecimal("8.99");
   stmt.addBatch("insert into COFFEES values " +
                "(2250, ’Colombian_Decaf’, 101, "+b+")");

            stmt.executeBatch();

            rs = stmt.executeQuery("select * from COFFEES");
            jdbcRs = new JdbcRowSetImpl(rs);

            jdbcRs.absolute(3);
            jdbcRs.updateBigDecimal("PRICE", new BigDecimal("9.99"));
            jdbcRs.updateRow();

            jdbcRs.first();
            jdbcRs.moveToInsertRow();
            jdbcRs.updateString("COF_NAME", "House_Blend");
            jdbcRs.updateInt("SUP_ID", 49);
            jdbcRs.updateBigDecimal("PRICE", new BigDecimal("7.99"));
            jdbcRs.insertRow();
            jdbcRs.moveToCurrentRow();

            jdbcRs.moveToInsertRow();
            jdbcRs.updateString("COF_NAME", "House_Blend_Decaf");
            jdbcRs.updateInt("SUP_ID", 49);
            jdbcRs.updateBigDecimal("PRICE", new BigDecimal("8.99"));
            jdbcRs.insertRow();
            jdbcRs.moveToCurrentRow();

            jdbcRs.last();
            jdbcRs.deleteRow();

            con.close();
            jdbcRs.close();

        } catch(SQLException sqle) {
            System.out.println("SQL Exception encountered: " + sqle.getMessage());
        }
    }
}
```

3 CachedRowSet

A CachedRowSet object is special in that it can operate without being connected to its data source, that is, it is a *disconnected* RowSet object. It gets the name “CachedRowSet” from the fact that it stores (caches) its data in memory so that it can operate on its own data rather than on the data stored in a database.

The CachedRowSet interface is the superinterface for all disconnected RowSet objects, so everything demonstrated in this chapter also applies to WebRowSet, JoinRowSet, and FilteredRowSet objects.

Note that although the data source for a CachedRowSet object (and the RowSet objects derived from it) is almost always a relational database, a CachedRowSet object is capable of getting data from any data source that stores its data in a tabular format. For example, a flat file or spread sheet could be the source of data. This is true when the RowSetReader object for a disconnected RowSet object is implemented to read data from such a data source. The reference implementation of the CachedRowSet interface has a RowSetReader object that reads data from a relational database, so in this tutorial, the data source is always a database.

Setting Up a CachedRowSet Object

Setting up a CachedRowSet object involves creating it, setting its properties, and setting its key columns.

Creating a CachedRowSet Object

You can create a new CachedRowSet object in two different ways:

- By using the default constructor
- By passing a SyncProvider implementation to the constructor

Using the Default Constructor

One of the ways you can create a CachedRowSet object is by calling the default constructor defined in the reference implementation, as is done in the following line of code.

```
CachedRowSet crs = new CachedRowSetImpl();
```

crs has the same default values for its properties that a JdbcRowSet object has when it is first created. In addition, it has been assigned an instance of the default SyncProvider implementation, RIOptimisticProvider.

A SyncProvider object supplies a RowSetReader object (a *reader*) and a RowSetWriter object (a *writer*), which a disconnected RowSet object needs in order to read data from its data source or to write data back to its data source. What a reader and writer do is explained later in the sections “What a Reader Does” (page 23) and “What a Writer Does” (page 26). One thing to keep in mind is that readers and writers work entirely behind the scenes, so the explanation of how they work is for your information only. Having some background on readers and writers should help you understand what some of the methods defined in the CachedRowSet interface do behind the scenes.

Passing a SyncProvider Implementation

A second way to create a CachedRowSet object is to pass the fully qualified name of a SyncProvider implementation to the CachedRowSetImpl constructor.

```
CachedRowSet crs2 = CachedRowSetImpl("com.fred.providers.HighAvailabilityProvider");
```

The preceding example assumes that com.fred.providers.HighAvailabilityProvider is a third party implementation of the SyncProvider interface. Presumably, this implementation has reader and writer implementations that differ from those in the RIOptimisticProvider implementation. You will see more about alternate possibilities, especially for writer implementations, later.

Setting Properties

Generally, the default values for properties are fine as they are, but you may change the value of a property by calling the appropriate setter method. And there are some properties without default values that you need to set yourself.

In order to get data, a disconnected RowSet object needs to be able to connect to a data source and have some means of selecting the data it is to hold. Four properties hold information necessary to obtain a connection to a database.

- username—the name a user supplies to a database as part of gaining access
- password—the user's database password
- url—the JDBC URL for the database to which the user wants to connect
- datasourceName—the name used to retrieve a DataSource object that has been registered with a JNDI naming service

As was mentioned in the chapter "Overview," which of these properties you need to set depends on how you are going to make a connection. The preferred way is to use a DataSource object, but it may not be practical for some readers to register a DataSource object with a JNDI naming service, which is generally done by a person acting in the capacity of a system administrator. Therefore, the code examples all use the DriverManager mechanism to obtain a connection, for which you use the url property and not the datasourceName property.

The following lines of code set the username, password, and url properties so that a connection can be obtained using the DriverManager mechanism. (You will find the JDBC URL to set as the value for the url property in the documentation for your JDBC driver.)

```
crs.setUsername("hardy");
crs.setPassword("oursecret");
crs.setUrl("jdbc:mySubprotocol:mySubname");
```

Another property that you must set is the command property. In the reference implementation, data is read into a RowSet object from a ResultSet object. The query that produces that ResultSet object is the value for the command property. For example, the following line of code sets the command property with a query that produces a ResultSet object containing all the data in the table COF_INVENTORY.

```
crs.setCommand("select * from COF_INVENTORY");
```

You will see how the command property is used and *crs* is filled with data later in this chapter.

Setting Key Columns

If you are going make any updates to *crs* and want those updates saved in the database, you must set one more piece of information: the key columns. Key columns are essentially the same as a primary key because they indicate one or more columns that uniquely identify a row. The difference is that a primary key is set on a table in the database, whereas key columns are set on a particular RowSet object. The following lines of code set the key columns for *crs* to the first column.

```
int [] keys = {1};
crs.setKeyColumns(keys);
```

The first column in the table COFFEES is COF_NAME. It can serve as the key column because every coffee name is different and therefore uniquely identifies one row and only one row in the table COFFEES. The method setKeyColumns takes an array to allow for the fact that it may take two or more columns to identify a row uniquely.

Key columns are used internally, so after setting them, you do nothing more with them. However, it is important that you set key columns because a SyncResolver object cannot identify specific rows without that information. You will see how to use a SyncResolver object in the section “Using a SyncResolver Object,” on page 27.

Populating a CachedRowSet Object

Populating a disconnected RowSet object involves more work than populating a connected RowSet object. The good news is that all the extra work is done behind the scenes; as a programmer, it is still very simple for you. After you have done the preliminary work to set up the CachedRowSet object *crs*, shown in the previous sections of this chapter, the following line of code populates *crs*.

```
crs.execute();
```

The data in *crs* is the data in the ResultSet object produced by executing the query in the command property.

What is different is that the CachedRowSet implementation for the method execute does a lot more than the JdbcRowSet implementation. Or more correctly, the

CachedRowSet object's reader, to which the method execute delegates its tasks, does a lot more.

Every disconnected RowSet object has a SyncProvider object assigned to it, and this SyncProvider object is what provides the RowSet object's *reader* (a RowSetReader object). When we created *crs*, we used the default CachedRowSetImpl constructor, which, in addition to setting default values for properties, assigns an instance of the RIOptimisticProvider implementation as the default SyncProvider object.

What a Reader Does

When an application calls the method execute, a disconnected RowSet object's reader works behind the scenes to populate the RowSet object with data. A newly created CachedRowSet object is not connected to a data source and therefore must obtain a connection to that data source in order to get data from it. The reference implementation of the default SyncProvider object (RIOptimisticProvider) provides a reader that obtains a connection by using the values set for the user name, password, and either the the JDBC URL or the data source name, whichever was set more recently. Then the reader executes the query set for the command. It reads the data in the ResultSet object produced by the query, populating the CachedRowSet object with that data. Finally, the reader closes the connection, making the CachedRowSet object lightweight again.

After the method execute has been called and the reader has populated the CachedRowSet object *crs* with the data from the table COF_INVENTORY, *crs* contains the data in Table 3–1.

Table 3–1 COF_INVENTORY

WAREHOUSE_ID	COF_NAME	SUP_ID	QUAN	DATE
1234	House_Blend	49	0	2006_04_01
1234	House_Blend_Decaf	49	0	2006_04_01
1234	Colombian	101	0	2006_04_01
1234	French_Roast	49	0	2006_04_01
1234	Espresso	150	0	2006_04_01

WAREHOUSE_ID	COF_NAME	SUP_ID	QUAN	DATE
1234	Colombian_Decaf	101	0	2006_04_01

Updating a CachedRowSet Object

In our ongoing Coffee Break scenario, the owner wants to streamline operations. He decides to have employees at the warehouse enter inventory directly into a PDA (personal digital assistant), thereby avoiding the error-prone process of having a second person do the data entry. A CachedRowSet object is ideal in this situation because it is lightweight, serializable, and can be updated without a connection to the data source.

The owner will have his programmer create a GUI tool for the PDA that his warehouse employees will use for entering inventory data. Headquarters will create a CachedRowSet object populated with the table showing the current inventory and send it via the Internet to the PDAs. When a warehouse employee enters data using the GUI tool, the tool adds each entry to an array, which the CachedRowSet object will use to perform the updates behind the scenes. Upon completion of the inventory, the PDAs send their new data back to headquarters, where the data is uploaded to the database server.

[???Is this how things would work? Please fix this as necessary.????? I know nothing about PDAs.]

Updating a Column Value

Updating data in a CachedRowSet object is just the same as updating data in a JdbcRowSet object. For example, the following code fragment could represent what a CachedRowSet object would do when a warehouse employee entered values to be set in the QUAN column of the table COF_INVENTORY. The date of the inventory was entered at headquarters, so that does not need to be changed. The cursor is moved to before the first row so that the first call to the method next will put the cursor on the first row. For each row, after the value for the column QUAN has been set with a new value, the method updateRow is called to save the new value to memory.

```
int [] quantity = {873, 927, 985, 482, 358, 531};
int len = quantity.length;
crs.beforeFirst();
while (crs.next()) {
    for(int i = 0; i < len; i++) {
        crs.updateInt("QUAN", quantity[i]);
        crs.updateRow();
    }
}
```

Inserting and Deleting Rows

Just as with updating a column value, the code for inserting and deleting rows in a CachedRowSet object is the same as for a JdbcRowSet object.

If the warehouse has received a shipment of a type of coffee that has not yet been entered in the COF_INVENTORY table, the GUI tool could have the warehouse employee enter the necessary information for adding a new row. The implementation of the tool could insert the new row into the CachedRowSet object *crs* with the following code fragment.

```
crs.moveToInsertRow();
crs.updateInt("WAREHOUSE_ID", 1234);
crs.updateString("COF_NAME", "Supremo");
crs.updateInt("SUP_ID", 150);
crs.updateInt("QUAN", 580);
java.util.Date 2006_04_01 = java.util.Date.valueOf("2006-04-01");
crs.updateDate("DATE", 2006_04_01);
crs.insertRow();
crs.moveToCurrentRow();
```

If headquarters has discontinued Espresso coffee, it would probably remove the row for that coffee itself. However, in our scenario, a warehouse employee using a PDA also has the capability of removing it. The following code fragment finds the row where the value in the COF_NAME column is Espresso and deletes it from *crs*.

```
while (crs.next()) {
    if (crs.getString("COF_NAME").equals("Espresso")) {
        crs.deleteRow();
        break;
    }
}
```

Updating the Data Source

There is a major difference between making changes to a JdbcRowSet object and making changes to a CachedRowSet object. Because a JdbcRowSet object is connected to its data source, the methods updateRow, insertRow, and deleteRow can update both the JdbcRowSet object and the data source. In the case of a disconnected RowSet object, however, these methods update the data stored in the CachedRowSet object's memory but cannot affect the data source. A disconnected RowSet object must call the method acceptChanges in order to save its changes to the data source. In our inventory scenario, back at headquarters, an application will call the method acceptChanges to update the database with the new values for the column QUAN.

```
crs.acceptChanges();
```

What a Writer Does

Like the method execute, the method acceptChanges does its work invisibly. Whereas the method execute delegates its work to the RowSet object's reader, the method acceptChanges delegates its tasks to the RowSet object's writer. Behind the scenes, the writer opens a connection to the database, updates the database with the changes made to the RowSet object, and then closes the connection.

Using the Default Implementation

The difficulty is that a *conflict* can arise. A conflict is a situation in which another party has updated a value in the database that corresponds to a value that was updated in a RowSet object. Which value should be persisted in the database? What the writer does when there is a conflict depends on how it is implemented, and there are many possibilities. At one end of the spectrum, the writer does not even check for conflicts and just writes all changes to the database. This is the case with the RIXMLProvider implementation, which is used by a WebRowSet object. At the other end, the writer makes sure there are no conflicts by setting database locks that prevent others from making changes.

The writer for *crs* is the one provided by the default SyncProvider implementation, RIOptimisticProvider. The RIOPtimisticProvider implementation gets its name from the fact that it uses an optimistic concurrency model. This model assumes that there will be few, if any, conflicts and therefore sets no database locks. The writer checks to see if there are any conflicts, and if there are none, it writes the changes made to *crs* to the database to be persisted. If there are any conflicts, the default is not to write the new RowSet values to the database.

In our scenario, the default behavior works very well. Because no one at headquarters is likely to change the value in the QUAN column of COF_INVENTORY, there will be no conflicts. As a result, the values entered into *crs* at the warehouse will be written to the database and thus persisted, which is the desired outcome.

Using a SyncResolver Object

In other situations, however, it is possible for conflicts to exist. To accommodate these situations, the RIOPtimisticProvider implementation provides an option that lets you look at the values in conflict and decide which ones to persist. This option is the use of a SyncResolver object. Keep in mind that you do not have to use a SyncResolver object; it is available as an option for those who want to be able to determine manually what changes are persisted.

When you call the method acceptChanges, one of the things the writer does is to check for conflicts. If it has found one or more, it creates a SyncResolver object containing the database values that caused the conflicts. In this case, the method acceptChanges throws a SyncProviderException object, which an application may catch and use to retrieve the SyncResolver object. The following lines of code retrieve the SyncResolver object *resolver*.

```
try {
    crs.acceptChanges();
} catch (SyncProviderException spe) {
    SyncResolver resolver = spe.getSyncResolver();
```

resolver is a RowSet object that replicates *crs* except that it contains only the values in the database that caused a conflict. All other column values are null.

With *resolver* in hand, you can iterate through its rows to locate the values that are not null and are therefore values that caused a conflict. Then you can locate the value at the same position in *crs* and compare them. The following code fragment retrieves *resolver* and uses the SyncResolver method nextConflict to iterate through the rows that have conflict values. *resolver* gets the status of each conflict value, and if it is UPDATE_ROW_CONFLICT, meaning that the *crs* was attempting an update when the conflict occurred, *resolver* gets the row number of that value. Then the code moves the cursor for *crs* to the same row. Next, the code finds the column in that row of *resolver* that contains a conflict value, which will be a value that is not null. After retrieving the value in that column from both *resolver* and *crs*, you can compare the two and decide which one you want to be persisted. Finally, the code sets that value in both *crs* and the database using the method setResolvedValue.

```
try {
    crs.acceptChanges();
} catch (SyncProviderException spe) {
    SyncResolver resolver = spe.getSyncResolver();

    Object crsValue; // value in crs
    Object resolverValue; // value in the SyncResolver object
    Object resolvedValue; // value to be persisted

    while (resolver.nextConflict()) {
        if (resolver.getStatus() == SyncResolver.UPDATE_ROW_CONFLICT) {
            int row = resolver.getRow();
            crs.absolute(row);

            int colCount = crs.getMetaData().getColumnCount();
            for (int j = 1; j <= colCount; j++) {
                if (resolver.getConflictValue(j) != null) {
                    crsValue = crs.getObject(j);
                    resolverValue = resolver.getConflictValue(j);
                    . . . // compare crsValue and resolverValue to determine the
                        // value to be persisted

                    resolvedValue = crsValue;
                    resolver.setResolvedValue(j, resolvedValue);
                }
            }
        }
    }
}
```

Note that the SyncResolver object uses key columns internally to identify specific rows. If you do not set the key column(s) (using the CachedRowSet method setKey-Columns) the SyncResolver object will not be able to function correctly.

Using Other SyncProvider Implementations

The JDBC RowSet Implementations provide two SyncProvider implementations: the RIOptimisticProvider, which is the default provider, and the RIXmlProvider, which a WebRowSet object uses. Developers are free to write their own implementations of the SyncProvider interface. The reader, for example, can be implemented to get its data from a data source other that a relational database. More likely, though, are variations in the behavior of the writer. Different writers can provide different levels of care in avoiding conflicts or different approaches to handling conflicts.

To make them available to others, developers register their SyncProvider implementations with the SyncFactory. You can find out what SyncProvider implementations are available by calling the SyncFactory.getRegisteredProviders method.

```
Enumeration providers = SyncFactory.getRegisteredProviders();
```

You can plug in an alternate provider simply by setting it as the provider. The following line of code, in which the argument is the fully qualified class name of a SyncProvider implementation, creates a CachedRowSet object initialized with the specified provider.

```
CachedRowSet crs = new CachedRowSetImpl(
                    "com.fred.providers.HighAvailablityProvider");
```

Another option is to change the provider after a CachedRowSet object has been created, as is done in the following line of code.

```
crs.setSyncProvider("com.fred.providers.HighAvailablityProvider");
```

Notifying Listeners

Being a JavaBeans component means that a RowSet object can notify other components when certain things happen to it. For example, if data in a RowSet object changes, the RowSet object can notify interested parties of that change. The nice thing about this notification mechanism is that, as an application programmer, all you have to do is add or remove the components that will be notified.

Setting Up Listeners

A *listener* for a RowSet object is a component that implements the following methods from the RowSetListener interface:

- cursorMoved—defines what the listener will do, if anything, when the cursor in the RowSet object moves
- rowChanged—defines what the listener will do, if anything, when one or more column values in a row have changed, a row has been inserted, or a row has been deleted
- rowSetChanged—defines what the listener will do, if anything, when the RowSet object has been populated with new data

An example of a component that might want to be a listener is a BarGraph object that graphs the data in a RowSet object. As the data changes, the BarGraph object can update itself to reflect the new data.

As an application programmer, the only thing you need to do to take advantage of the notification mechansim is to add or remove listeners. The following line of code means that every time the cursor for *crs* moves, values in *crs* are changed, or *crs* as a whole gets new data, the BarGraph object *bar* will be notified.

```
crs.addRowSetListener(bar);
```

You can also stop notifications by removing a listener, as is done in the following line of code.

```
crs.removeRowSetListener(bar);
```

In our Coffee Break scenario, let’s assume that headquarters checks with the database periodically to get the latest price list for the coffees it sells online. In this case, the listener is the PriceList object *priceList* at the Coffee Break web site, which must implement the RowSetListener methods cursorMoved, rowChanged, and rowSetChanged. The implementation of cursorMoved could be to do nothing because the position of the cursor does not affect *priceList*. The implementations for rowChanged and rowSetChanged, on the other hand, need to specify what is to be done to update *priceList*. Because the listener in this case is part of a Web service, the implementations will probably send the latest data in a RowSet object in XML format, which is effectively the standard format for Web services communications. The chapter “WebRowSet,” starting on page 77, shows an easy way to send data in XML format.

How Notification Works

In the reference implementation, methods that cause any of the RowSet events automatically notify all registered listeners. For example, any method that moves the cursor also calls the method cursorMoved on each of the listeners. Similarly, the method execute calls the method rowSetChanged on all listeners, and acceptChanges calls rowChanged on all listeners.

Accessing Large Amounts of Data

The Coffee Break chain has expanded into selling all kinds of coffee-related merchandise and sends a catalog of merchandise to all of its coffee houses so that the managers can order what is appropriate for a particular location. This catalog is in the form of a database table with hundreds of rows. The owner wants to send the catalog in the form of a CachedRowSet object but is worried that it may be too big.

A CachedRowSet object, like all disconnected RowSet objects, stores its data in memory; therefore, the amount of data it can hold is limited by the size of its memory. But by using *paging*, a CachedRowSet object can handle amounts of data that exceed its memory limit. Paging involves getting data from a ResultSet object in chunks of data called *pages*. If you have set the size of a page at 100, for example, you will get up to 100 rows of data in your CachedRowSet object at any one time. The following line of code sets the page size for *crs* to 100, meaning that data will be fetched in chunks of 100 rows at a time.

```
crs.setPageSize(100);
```

After setting properties and setting the page size, you call the method execute or populate. Because the page size has been set to 100, the method execute, used in the following line of code, executes the command for *crs* and populates *crs* with the first 100 rows from the resulting ResultSet object.

```
crs.execute();
```

The method for getting subsequent rows is nextPage, which increments the current page of *crs*, fetches the next 100 rows, and reads them into *crs*. You can use the method nextPage in a while loop to get all of the rows because it will keep fetching 100 rows at a time until there are no more rows, at which time nextPage returns false and ends the loop. The code fragment that follows uses a second while loop within the first one, which uses the method next to iterate through each row of each page.

If, for example, you want to update the quantity for item 1235, you need to do the work within the inner while loop to be sure that you will find the row where item 1235 is located. The following code iterates through each page until it finds item 1235 and then updates its quantity. The code then calls the method updateRow to save the update to memory and the method acceptChanges to save the update to the database.

```
crs.setPageSize(50);
crs.execute();
while(crs.next()) {
    if (crs.getInt("ITEM_ID") == 1235) {
        System.out.println("QUAN value: " + crs.getInt("QUAN"));
        crs.updateInt("QUAN", 99);
        crs.updateRow();

while(crs.nextPage()) {
    System.out.println("Page number: " + i);
    while(crs.next()) {
        if (crs.getInt("ITEM_ID") == 1235) {
            System.out.println("QUAN value: " + crs.getInt("QUAN"));
            crs.updateInt("QUAN", 99);
            crs.updateRow();
            crs.acceptChanges();
        }
    }
    i++;
}
crs.acceptChanges();
```

If you have reached the end of the data and want to go back through it in reverse, you can use the method previousPage. This method decrements the number of the current page and fetches the previous 50 rows (or whatever number the page size is). You can go back through all the pages by putting previousPage in a while loop, analogous to going forward through all the pages with the method nextPage in a while loop.

Code Sample

This sample code demonstrates paging and using the method acceptChanges. Headquarters is sending an updated list of all the merchandise that individual Coffee Break coffee houses can order. Because it is presumably very large, we will not use the entire table in the example. For example purposes, we will use only twelve rows and set the page size to 4, which means that there will be three pages. The sample code does the following:

1. Creates the table MERCH_CATALOG and inserts data into it.The data types for the columns in the table MERCH_CATALOG are:
   ITEM_ID—INTEGER
   ITEM_NAME—VARCHAR(20)
   SUP_ID—INTEGER
   PRICE—DECIMAL(6,2)

   Note that instead of using SQL INSERT statements, data is inserted into the table programmatically. That is, after calling the method moveToInsertRow, updater methods and insertRow are called to insert a row of data.

Table 3–2 MERCH_CATALOG

ITEM_ID	ITEM_NAME	SUP_ID	PRICE
00001234	Cup_Large	00456	5.99
00001235	Cup_Small	00456	2.99
00001236	Saucer	00456	2.99
00001287	Carafe	00456	25.99
00006931	Carafe	00927	44.99
00006935	PotHolder	00927	3.50
00006977	Napkin	00927	3.99
00006979	Towel	00927	4.99
00004488	CofMaker	08732	89.99
00004490	CofGrinder	08732	59.99

ITEM_ID	ITEM_NAME	SUP_ID	PRICE
00004495	EspMaker	08732	79.99
00006914	Cookbook	00927	15.00

2. Creates a CachedRowSet object and sets its properties so that it can make a connection to the database.

3. Populates the CachedRowSet object using the method execute. Uses paging to send four rows at a time, which will require three CachedRowSet objects.

4. Updates the price of the small cup (ITEM_ID 1235) to: new BigDecimal("3.50").

5. Adds a new row for a new item. Values (0006914, "Tablecloth", 00927, new BigDecimal("19.99").

6. Calls the method acceptChanges to update the database with the changes made in 4 and 5.

==

```
import java.sql.*;
import javax.sql.rowset.*;
import java.math.BigDecimal;
import com.sun.rowset.*;

public class CachedRowSetSample {

    public static void main( String [] args) {

        String strUrl = "jdbc:datadirect:oracle://" +
                    "129.158.229.21:1521;SID=ORCL9";
        String strUserId = "scott";
        String strPassword = "tiger";
        String className = "com.ddtek.jdbc.oracle.OracleDriver";
        CachedRowSet crs;
        int i = 1;

        try {
            Class.forName(className);
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage);
        }
```

```
      try {
          Connection con = DriverManager.getConnection(
                                       strUrl, strUserId, strPassword);
          con.setAutoCommit(false);

          Statement stmt = con.createStatement();
          stmt.executeUpdate("drop table MERCH_INVENTORY");
          stmt.executeUpdate("create table MERCH_INVENTORY( " +
              "ITEM_ID INTEGER, ITEM_NAME VARCHAR(20), " +
              "SUP_ID INTEGER, PRICE DECIMAL(6,2))");
          PreparedStatement pStmt = con.prepareStatement(
                  "insert into MERCH_INVENTORY values(?, ?, ?, ?)");

// inserting values for 12 rows
          pStmt.setInt(1, 1234);
          pStmt.setString(2, "Cup_Large");
          pStmt.setInt(3, 456);
          pStmt.setBigDecimal(4, new BigDecimal("5.99"));
          pStmt.executeUpdate();

          pStmt.setInt(1, 1235);
          pStmt.setString(2, "Cup_Small");
          pStmt.setInt(3, 456);
          pStmt.setBigDecimal(4, new BigDecimal("2.99"));
          pStmt.executeUpdate();

          pStmt.setInt(1, 1236);
          pStmt.setString(2, "Saucer");
          pStmt.setInt(3, 456);
          pStmt.setBigDecimal(4, new BigDecimal("2.99"));
          pStmt.executeUpdate();

          pStmt.setInt(1, 1287);
          pStmt.setString(2, "Carafe");
          pStmt.setInt(3, 456);
          pStmt.setBigDecimal(4, new BigDecimal("25.99"));
          pStmt.executeUpdate();

          pStmt.setInt(1, 6931);
          pStmt.setString(2, "Carafe");
          pStmt.setInt(3, 927);
          pStmt.setBigDecimal(4, new BigDecimal("44.99"));
          pStmt.executeUpdate();

          pStmt.setInt(1, 6935);
          pStmt.setString(2, "PotHolder");
          pStmt.setInt(3, 927);
pStmt.setBigDecimal(4, new BigDecimal("3.50"));
pStmt.executeUpdate();

pStmt.setInt(1, 6977);
pStmt.setString(2, "Napkin");
pStmt.setInt(3, 927);
pStmt.setBigDecimal(4, new BigDecimal("3.99"));
pStmt.executeUpdate();

pStmt.setInt(1,6979);
pStmt.setString(2, "Towel");
pStmt.setInt(3, 927);
pStmt.setBigDecimal(4, new BigDecimal("4.99"));
pStmt.executeUpdate();

pStmt.setInt(1, 4488);
pStmt.setString(2, "CofMaker");
pStmt.setInt(3, 8372);
pStmt.setBigDecimal(4,new BigDecimal("89.99"));
pStmt.executeUpdate();

pStmt.setInt(1, 4490);
pStmt.setString(2, "CofGrinder");
pStmt.setInt(3, 8732);
pStmt.setBigDecimal(4, new BigDecimal("59.99"));
pStmt.executeUpdate();

pStmt.setInt(1, 4495);
pStmt.setString(2, "EspMaker");
pStmt.setInt(3, 8732);
pStmt.setBigDecimal(4, new BigDecimal("79.99"));
pStmt.executeUpdate();

pStmt.setInt(1, 6914);
pStmt.setString(2, "Cookbook");
pStmt.setInt(3, 927);
pStmt.setBigDecimal(4, new BigDecimal("15.00"));
pStmt.executeUpdate();

con.commit();
con.close();

crs = new CachedRowSetImpl();

crs.setUrl(strUrl);
crs.setUsername(strUserId);
crs.setPassword(strPassword);
crs.setCommand("select * from MERCH_CATALOG");
```

```
// Setting the page size to 4, such that we get the data
// in chunks of 4 rows at a time
            crs.setPageSize(4);

// Get the first set of data
            crs.execute();
            while(crs.next()) {
                if (crs.getInt("ITEM_ID") == 1235) {
                    System.out.println("PRICE value: "
                        + crs.getBigDecimal("PRICE"));
                    crs.updateBigDecimal("PRICE",
                                        new BigDecimal("3.50"));
                    crs.updateRow();
                    System.out.print("New PRICE value: ");
                    System.out.println(crs.getBigDecimal("PRICE"));
                }
            }

// Keep on getting data in chunks until done
            while(crs.nextPage()) {
                System.out.println("Page number: "+i);
                while(crs.next()) {
                    if(crs.getInt("ITEM_ID") == 1235) {
                        System.out.print("PRICE value: ");
                        System.out.println(crs.getBigDecimal("PRICE"));
                        crs.updateBigDecimal("PRICE", new BigDecimal("3.50"));
                        crs.updateRow();
                        System.out.print("New PRICE value: ");
                        System.out.println(crs.getBigDecimal("PRICE"));
                    }
                } // End of inner while
                i++;
            } // End of outer while

// Inserting a new row
// Calling previousPage to come back to the last page
// as we'll be after the last page.
            crs.previousPage();
            crs.moveToInsertRow();
            crs.updateInt("ITEM_ID", 6922);
            crs.updateString("ITEM_NAME", "TableCloth");
            crs.updateInt("SUP_ID", 927);
            crs.updateBigDecimal("PRICE", new BigDecimal("19.99"));
            crs.insertRow();
            crs.moveToCurrentRow();

    // Synchronizing the update and new row back to the database
```

```
            crs.acceptChanges();

            crs.close();

        } catch( SQLException sqle) {
            System.out.println("SQLException caught: " + sqle.getMessage());
        }
    } // End of main
} // End of class
```

4

JoinRowSet

A JoinRowSet implementation lets you create an SQL JOIN between RowSet objects when they are not connected to a data source. This is important because it saves the overhead of having to create one or more connections. In this chapter, you will learn how to do the following:

- Create a JoinRowSet object
- Add RowSet objects to form an SQL JOIN
- Manage match columns

NOTE: You should have read the chapter “CachedRowSet” before reading this chapter. The JoinRowSet interface is a subinterface of the CachedRowSet interface and thereby inherits the capabilities of a CachedRowSet object. This means that a JoinRowSet object is a disconnected RowSet object and can operate without always being connected to a data source.

Creating a JoinRowSet Object

A JoinRowSet object serves as the holder of an SQL JOIN. The following line of code shows how easy it is to create a JoinRowSet object.

```
JoinRowSet jrs = new JoinRowSetImpl();
```

The variable *jrs* has the default properties that a default CachedRowSet object has, but it has no data until RowSet objects are added to it.

Adding RowSet Objects

Any RowSet object can be added to a JoinRowSet object as long as it can be part of an SQL JOIN. A JdbcRowSet object, which is always connected to its data source, can be added, but normally it forms part of a JOIN by operating with the data source directly instead of becoming part of a JOIN by being added to a JoinRowSet object. The point of providing a JoinRowSet implementation is to make it possible for disconnected RowSet objects to become part of a JOIN relationship.

The owner of The Coffee Break chain of coffee houses wants to get a list of the coffees he buys from Acme, Inc. In order to do this, he will have to get information from two tables, COFFEES and SUPPLIERS. In the database world before RowSet technology, the owner or his programmers would send the following query to the database:

```
String query = "SELECT COFFEES.COF_NAME FROM COFFEES, SUPPLIERS " +
               "WHERE SUPPLIERS.SUP_NAME =  Acme.Inc.  and " +
               "SUPPLIERS.SUP_ID = COFFEES.SUP_ID";
```

In the world of RowSet technology, you can accomplish the same result without having to send a query to the data source. You can add RowSet objects containing the data in the two tables to a JoinRowSet object. Then, because all the pertinent data is in the JoinRowSet object, you can perform a query on it to get the desired data.

The following code fragments create two CachedRowSet objects, *coffees* populated with the data from the table COFFEES, and *suppliers* populated with the data from the table SUPPLIERS. The readers (RowSetReader objects) for *coffees* and *suppliers* have to make a connection to the database to execute their commands and get populated with data, but once that is done, they do not have to reconnect again in order to form a JOIN. You can form any number of JOIN relationships from these two RowSet objects using *jrs* while it is disconnected.

```
CachedRowSet coffees = new CachedRowSetImpl();
coffees.setCommand("SELECT * FROM COFFEES");
coffees.setUsername(name);
coffees.setPassword(password);
coffees.setURL("jdbcDriverURL1");
coffees.execute();

CachedRowSet suppliers = new CachedRowSetImpl();
suppliers.setCommand("SELECT * FROM SUPPLIERS");
suppliers.setUsername(name);
suppliers.setPassword(password);
suppliers.setURL("jdbcDriverURL2");
suppliers.execute();
```

The contents of these two tables are shown in Table 4–1 and Table 4–2. Note that for this example, the tables have different columns than in some other examples to make displaying the JOIN result (Table 4–3 on page 48) more manageable..

Table 4–1 COFFEES

COF_ID	**COF_NAME**	**SUP_ID**	**PRICE**
1250	Colombian	101	7.99
1300	French_Roast	49	8.99
1800	Espresso	150	10.99
2250	Colombian_Decaf	101	8.99
1000	House_Blend	49	7.99
2000	House_Blend_Decaf	49	8.99

Table 4–2 SUPPLIERS

SUP_ID	**SUP_NAME**	**ADDRESS**
101	Acme, Inc.	Groundsville1
49	Superior Coffee	Mendocino1

Table 4–2 SUPPLIERS

SUP_ID	SUP_NAME	ADDRESS
150	The High Ground	Meadows1

Looking at the SUPPLIERS table, you can see that Acme, Inc. has an identification number of 101. The coffees in the table COFFEES with the supplier identification number of 101 are Colombian and Colombian_Decaf. The joining of information from both tables is possible because the two tables have the column SUP_ID in common. In JDBC RowSet technology, SUP_ID, the column on which the JOIN is based, is called the *match column*.

Each RowSet object added to a JoinRowSet object must have a match column, the column on which the JOIN is based. There are two ways to set the match column for a RowSet object. The first way is to pass the match column to the JoinRowSet method addRowSet. The second way is to use the Joinable method setMatchColumn.

Passing the Match Column to addRowSet

A RowSet object added to a JoinRowSet object must have a column that matches a column in all of the other RowSet objects in the JoinRowSet object. For example, the tables *coffees* and *suppliers* both have the column SUP_ID, so that is the match column for these two tables. If another table is added to the JoinRowSet object, it, too, must have a SUP_ID column.

The following line of code passes the method addRowSet two arguments: the CachedRowSet object *coffees* and the column number of the match column. This method adds *coffees* to *jrs* and sets the match column for *coffees* to 3, which is the column SUP_ID.

```
jrs.addRowSet(coffees, 3);
```

coffees, being the first RowSet object to be added to *jrs*, now forms the basis for the JOIN relationship, meaning that any RowSet object added to *jrs* must have SUP_ID as its match column.

You can pass the method addRowSet the column name of the match column rather than the column number if you like.

```
jrs.addRowSet(coffees, "SUP_ID");
```

The next RowSet object added to *jrs* is *suppliers*, which can be added because it also has the column SUP_ID. The following line of code adds *suppliers* to *jrs* and sets the column SUP_ID as the match column.

```
jrs.addRowSet(suppliers, 1);
or
jrs.addRowSet(suppliers, "SUP_ID");
```

Using Joinable.setMatchColumn

A second way to set a match column is to use the Joinable method setMatchColumn. All RowSet interfaces are subinterfaces of the Joinable interface, meaning that they implement the Joinable interface. Accordingly, all five of the RowSet implementations implement the Joinable interface. Therefore, any RowSet object created from one of the constructors in the reference implementation has the ability to call Joinable methods. The following lines of code, in which *crs* is a CachedRowSet object, set the match column for *crs* and then add it to the JoinRowSet object *jrs*.

```
(Joinable)crs.setMatchColumn(1);
JoinRowSet jrs = new JoinRowSetImpl();
jrs.addRowSet(crs);
```

The Joinable interface provides methods for setting match columns, getting match columns, and unsetting match columns.

Using Multiple Columns as the MatchColumn

It is possible to have two or more columns serve as the match necessary for a JOIN relationship. In this case, the columns are elements in an array of int values or an array of String objects. The following lines of code demonstrate creating an array of column indexes and setting that array as the match column. When the match column has already been set, the method addRowSet takes only the RowSet object to be added as an argument.

```
int [] matchCols = {1, 3};
crs.setMatchColumn(matchCols);
jrs.addRowSet(crs);
```

The following two lines define an array of String objects and set that array as the match column.

```
String [] matchCols = {"SUP_ID", "ADDRESS"};
crs.setMatchColumn(matchCols);
jrs.addRowSet(crs);
```

You can also use the method addRowSet to set multiple columns as the match column when no match column has been set previously.

```
int [] matchCols = {1, 3};
jrs.addRowSet(crs, matchCols);
or
String [] matchCols = {"SUP_ID", "ADDRESS"};
jrs.addRowSet(crs, matchCols);
```

Using a JoinRowSet Object

Now *jrs* contains a JOIN between *coffees* and *suppliers* from which the owner can get the names of the coffees that he buys from Acme, Inc. Because the code did not set the type of JOIN, *jrs* holds an inner JOIN, which is the default. In other words, a row in *jrs* combines a row in *coffees* and a row in *suppliers*. It holds the columns in *coffees* plus the columns in *suppliers* for rows in which the value in the COFFEES.SUP_ID column matches the value in SUPPLIERS.SUP_ID. The following code prints out the names of coffees supplied by Acme, Inc. Note that this is possible because the column SUP_NAME, which is from *suppliers*, and COF_NAME, which is from *coffees*, are now both included in the JoinRowSet object *jrs*.

```
System.out.println("Coffees bought from Acme, Inc.: ");
while (jrs.next()) {
    if (jrs.getString("SUP_NAME").equals("Acme, Inc.")) {
        String name = jrs.getString("COF_NAME");
        System.out.println("        " + name);
    }
}
```

This will produce output similar to the following:

```
Coffees bought from Acme, Inc.:
        Colombian
        Colombian_Decaf
```

The JoinRowSet interface provides constants for setting the type of JOIN that will be formed, but currently the only type that is implemented is JoinRowSet.INNER_JOIN.

Code Sample

The following code sample combines code from throughout the chapter into a program you can run after you substitute the appropriate information for the variables *url*, *userId*, *passWord*, and *strDriver*.

This sample code demonstrates using a JoinRowSet object to perform a JOIN on the tables COFFEES and SUPPLIERS based on SUP_ID as the match column. This code sets the match column by supplying the column name to the method addRowSet.

==

```
import java.sql.*;
import javax.sql.rowset.*;
import com.sun.rowset.*;
import java.math.BigDecimal;

public class JoinRowSetSample {
    public static void main(String []args) {

        String strUrl = "jdbc:datadirect:oracle://" +
                    "129.158.229.21:1521;SID=ORCL9";
        String strUserId = "scott";
        tring strPassword = "tiger";
        String className = "com.ddtek.jdbc.oracle.OracleDriver";
        BigDecimal b;

        try {
            Class.forName(className);
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
```

```
Connection con = DriverManager.getConnection(
                              strUrll, strUserId, strPassWord);
con.setAutoCommit(false);

Statement stmt = con.createStatement();
stmt.addBatch("drop table COFFEES");
stmt.addBatch("drop table SUPPLIERS");

stmt.addBatch("create table COFFEES(COF_ID INTEGER, " +
          "COF_NAME VARCHAR(20), SUP_ID INTEGER, " +
          "PRICE DECIMAL(6,2))");

b = new BigDecimal("7.99");
stmt.addBatch("insert into COFFEES values " +
          "(1250, 'Colombian', 101, "+b+")");

b = new BigDecimal("8.99");
stmt.addBatch("insert into COFFEES values " +
          "(1300, 'French_Roast', 49, "+b+")");

b = new BigDecimal("10.99");
stmt.addBatch("insert into COFFEES values " +
          "(1800, 'Espresso', 150, "+b+")");

b = new BigDecimal("8.99");
stmt.addBatch("insert into COFFEES values " +
     "(2250, 'Colombian_Decaf', 101, "+b+")");

b = new BigDecimal("7.99");
stmt.addBatch("insert into COFFEES values " +
     "(1000, 'House_Blend', 491, "+b+")");

b = new BigDecimal("8.99");
stmt.addBatch("insert into COFFEES values " +
     "(2000, 'House_Blend_Decaf', 49, "+b+"");

stmt.addBatch("create table SUPPLIERS" +
     "(SUP_ID INTEGER, SUP_NAME VARCHAR(40), " +
     "ADDRESS VARCHAR(60))");

stmt.addBatch("insert into SUPPLIERS values " +
          "(101, 'Acme, Inc.', 'Groundsville, CA 95199'");

stmt.addBatch("insert into SUPPLIERS values " +
     "(49, 'Superior Coffee', 'Mendocino, CA 95460'");

stmt.addBatch("insert into SUPPLIERS values " +
     "(150, 'The High Ground', 'Meadows, CA 93966'");
```

```
            stmt.executeBatch();

            con.commit();

            ResultSet rs1 = stmt.executeQuery(
                                "select * from COFFEES");
            ResultSet rs2 = stmt.executeQuery(
                                "select * from SUPPLIERS");

// Populate two CachedRowSet objects and add them to a JoinRowSet object
            CachedRowSet coffees = new CachedRowSetImpl();
            coffees.populate(rs1);
            System.out.print("First CachedRowSet size: ");
            System .out.println(coffees.size());

            CachedRowSet suppliers = new CachedRowSetImpl();
            suppliers.populate(rs2);
            System.out.print("Second CachedRowSet size: ");
            System .out.println(suppliers.size());

            con.close();

            JoinRowSet jrs = new JoinRowSetImpl();
            jrs.addRowSet(coffees, "SUP_ID");
            jrs.addRowSet(suppliers, "SUP_ID");
            System.out.print("Size of the JoinRowSet is: ");
            System .out.println(jrs.size());
            System.out.println("Contents are ");
            while(jrs.next()) {
                System.out.print("COF_ID: ");
                System.out.println( jrs.getInt("COF_ID"));
                System.out.print("COF_NAME: " );
                System.out.println( jrs.getString("COF_NAME"));
                System.out.print("PRICE: " );
                System.out.println(jrs.getGetBigDecimal("PRICE"));
                System.out.print("SUP_ID: " );
                System.out.println( jrs.getInt("SUP_ID"));
                System.out.print("SUP_NAME: ");
                System.out.println( jrs.getString("SUP_NAME"));
                System.out.print("ADDRESS: ");
                System.out.println(jrs.getString("ADDRESS"));
            }
            jrs.close();
        } catch(SQLException sqle) {
            System.out.println("Caught SQLException: "
                                   + sqle.getMessage());
                    }
               }
          }
```

Table 4–3 represents the results of the inner JOIN of the tables COFFEES and SUPPLIERS.

Table 4–3 Inner JOIN of COFFEES and SUPPLIERS

COF_ID	COF_NAME	PRICE	SUP_ID	SUP_NAME	ADDRESS
1250	Colombian	7.99	101	Acme, Inc.	Groundsville, CA 95199
1300	French_Roast	8.99	49	Superior Coffee	Mendocino, CA 95460
1800	Espresso	10.99	150	The High Ground	Meadows, CA 93966
2250	Colombian_Decaf	8.99	101	Acme, Inc.	Groundsville, CA 95199
1000	House_Blend	7.99	49	Superior Coffee	Mendocino, CA 95460
2000	House_Blend_Decaf	8.99	49	Superior Coffee	Mendocino, CA 95460

5
FilteredRowSet

A FilteredRowSet object lets you cut down the number of rows that are visible in a RowSet object so that you can work with only the data that is relevant to what you are doing. You decide what limits you want to set on your data (how you want to “filter” the data) and apply that filter to a FilteredRowSet object. In other words, the FilteredRowSet object makes visible only the rows of data that fit within the limits you set. A JdbcRowSet object, which always has a connection to its data source, can do this filtering with a query to the data source that selects only the columns and rows you want to see. The query’s WHERE clause defines the filtering criteria. A FilteredRowSet object provides a way for a disconnected RowSet object to do this filtering without having to execute a query on the data source, thus avoiding having to get a connection to the data source and sending queries to it.

For example, assume that the Coffee Break chain of coffee houses has grown to hundreds of stores throughout the country, and all of them are listed in a table called COFFEE_HOUSES. The owner wants to measure the success of only the stores in California using his laptop, which cannot make a connection to his database system. This comparison will look at the profitability of selling merchandise versus selling coffee drinks plus various other measures of success, and it will rank California stores by coffee drink sales, merchandise sales, and total sales. Because the table COFFEE_HOUSES has hundreds of rows, these comparisons will be faster and easier if the amount of data being searched is cut down to only those rows where the value in the column STORE_ID indicates California.

This is exactly the kind of problem that a FilteredRowSet object addresses by providing the following capabilities:

- Ability to limit the rows that are visible according to set criteria
- Ability to select which data is visible without being connected to a data source

In this chapter you will walk through how to do the following:

- Define filtering criteria in a Predicate object
- Create a FilteredRowSet object and set it with a Predicate object
- Set a FilteredRowSet object with a second Predicate object to filter data even further
- Update a FilteredRowSet object
- Remove all filters so that all rows are once again visible

Creating a Predicate Object

To set the criteria for which rows in a FilteredRowSet object will be visible, you define a class that implements the Predicate interface. This Predicate object is initialized with the following:

- The high end of the range within which values must fall
- The low end of the range within which values must fall
- The column name or column number of the value that must fall within the range of values set by the high and low boundaries

Note that the range of values is inclusive, meaning that a value at the boundary is included in the range. For example, if the range has a high of 100 and a low of 50, a value of 50 is considered to be within the range. A value of 49 is not. Likewise, 100 is within the range, but 101 is not.

In line with the scenario where the owner wants to compare California stores, we need to write an implementation of the Predicate interface that filters for Coffee Break coffee houses located in California. There is no one right way to do this, which means that we have a lot of latitude in how we write the implementation. For example, we are free to name the class and its members whatever we want and to implement a constructor and the three evaluate methods in any way that accomplishes what we want.

The table listing all of the coffee houses, named COFFEE_HOUSES, has hundreds of rows. To make things more manageable, only part of the table is shown in this example, but it is enough so that you can see how it is set up and how the filtering is done.

A value in the column STORE_ID is an int that indicates, among other things, the state in which the coffee house is located. A value beginning with 10, for example, means that the state is California. STORE_ID values beginning with 32 indicate Oregon, and those beginning with 33 indicate the state of Washington. Table 5–1 shows an abbreviated version of the table COFFEE_HOUSES.

Table 5–1 COFFEE_HOUSES

STORE_ID	CITY	COFFEE	MERCH	TOTAL
10023	Mendocino	3450.55	2005.21	5455.76
33002	Seattle	4699.39	3109.03	7808.42
10040	SF	5386.95	2841.27	8228.22
32001	Portland	3147.12	3579.52	6726.64
10042	SF	2863.35	1874.62	4710.97
10024	Sacramento	1987.77	2341.21	4328.98
10039	Carmel	2691.69	1121.21	3812.90
10041	LA	1533.48	1007.02	2540.50
33002	Olympia	2733.83	1550.48	4284.31
33010	Seattle	3210.22	2177.48	5387.70
10035	SF	1922.85	1056.91	2979.76.
10037	LA	2143.38	1876.66	4020.04
10034	San_Jose	1234.55	1032.99	2267.54
32004	Eugene	1356.03	1112.81	2468.84
10041	LA	2284.46	1732.97	4017.43

Our implementation could look like the following.

```
public class Filter1 implements Predicate {
    private int lo;
    private int hi;
    private String colName;
    private int colNumber;


    public Filter1(int lo, int hi, int colNumber) {
        this.lo = lo;
        this.hi = hi;
        this.colNumber = colNumber;
    }

    public Filter1(int lo, int hi, String colName) {
        this.lo = lo;
        this.hi = hi;
        this.colName = colName;
    }

    public boolean evaluate(RowSet rowset) {
        CachedRowSet crs = (CachedRowSet)rowset;

        if (rowset.getInt(colNumber) >= lo &&
                (rowset.getInt(colNumber) <= hi) {
            return true;
        } else {
            return false;
        }
    }
```

This is a very simple implementation that checks the value in the column STORE_ID in the current row of the given RowSet object to see if it is in the prescribed range.

The following line of code creates a Filter1 object that tests whether the value in the column STORE_ID is within the range of 10000 to 10999, inclusive.

```
Filter1 range = new Filter1(10000, 10999, "STORE_ID");
```

All Coffee Break coffee houses in California have an identification number in which the first two digits of a five-digit number are 10, so the Filter1 object *range* allows only the rows where the STORE_ID column value indicates a California location to be visible.

Note that the Filter1 object just defined applies to one column. It is possible to have it apply to two or more columns by making each of the parameters arrays instead of single values. For example, the constructor for a Filter object could look like the following:

```
public Filter2(Object [] lo, Object [] hi, Object [] colNumber) {
    this.lo = lo;
    this.hi = hi;
    this.colNumber = colNumber;
}
```

The first element in *colNumber* gives the first column in which the value will be checked against the first element in *lo* and the first element in *hi*. The value in the second column indicated by *colNumber* will be checked against the second elements in *lo* and *hi*, and so on. Therefore, the number of elements in the three arrays should be the same. Here is what an implementation of the method evaluate(RowSet rs) might look like for a Filter2 object, in which the parameters are arrays.

```
public boolean evaluate(RowSet rs) {
    CachedRowSet crs = (CachedRowSet)rs;
    boolean bool1 = true;
    boolean bool2 = false;

    for (int i = 0; i < colNumber.length; i++) {
        if ((rs.getObject(colNumber[i] >= lo [i]) &&
        (rs.getObject(colNumber[i] <= hi[i]) {
        return bool1;
    } else {
        return bool2;
    }
}
```

The advantage of using a Filter2 implementation is that you can use parameters of any Object type and can check one column or multiple columns without having to write another implementation. However, you must pass an Object type, which means that you must convert a primitive type to its Object type. For example, if you use an int for *lo* and *hi*, you must convert the int to an Integer object before passing it to the constructor. String objects are already an Object type, so you do not have to convert them. The following line of code creates an Integer object that you could pass to the constructor for a Filter2 object.

```
Integer loInt = new Integer("10000");
```

Creating a FilteredRowSet Object

The reference implementation for the FilteredRowSet interface, FilteredRowSetImpl, includes a default constructor, which is used in the following line of code to create the empty FilteredRowSet object *frs*.

```
FilteredRowSet frs = new FilteredRowSetImpl();
```

The implementation extends the BaseRowSet abstract class, so *frs* has the default properties defined in BaseRowSet. This means that *frs* is scrollable, updatable, does not show deleted rows, has escape processing turned on, and so on. Also, because the FilteredRowSet interface is a subinterface of CachedRowSet, Joinable, and WebRowSet, *frs* has the capabilities of each. It can operate as a disconnected RowSet object, can be part of a JoinRowSet object, and can read and write itself in XML format using the RIXmlProvider.

Like other disconnected RowSet objects, *frs* must populate itself with data from a tabular data source, which is a relational database in the reference implementation. The following code fragment sets the properties necessary to connect to a database to execute its command. As pointed out earlier, code in this tutorial uses the DriverManager class to make a connection, which is done for convenience. Normally, it is better to use a DataSource object that has been registered with a naming service that implements the Java™ Naming and Directory Interface (JNDI).

```
frs.setCommand("SELECT * FROM COFFEE_HOUSES");
frs.setURL("jdbc:mySubprotocol:myDatabase");
frs.setUsername("Vladimir");
frs.setPassword("secret");
```

The following line of code populates *frs* with the data stored in the COFFEE_HOUSE table.

```
frs.execute();
```

Remember that the method execute does all kinds of things behind the scenes by calling on the RowSetReader object for *frs*, which creates a connection, executes the command for *frs*, populates *frs* with the data from the ResultSet object that is produced, and closes the connection. Note that if the table COFFEE_HOUSES had more rows than *frs* could hold in memory at one time, the CachedRowSet paging methods would have been used. If you need a refresher on paging or the method execute, see the chapter “CachedRowSet,” starting on page 19.

In our scenario, the Coffee Break owner would have done the preceding tasks in the office and then downloaded *frs* to his laptop. From now on, *frs* will operate independently, without the benefit of a connection to the data source.

Creating and Setting a Predicate Object

Now that the FilteredRowSet object *frs* contains the list of Coffee Break establishments, we can set selection criteria for narrowing down the number of rows in *frs* that are visible. The following line of code uses the Filter1 class we defined earlier in the chapter to create the object *range*, which checks the column STORE_ID to determine which stores are in California (which STORE_ID numbers fall between 10000 and 10999, inclusive).

```
Filter1 range = new Filter1(10000, 10999, "STORE_ID");
```

The next line of code sets *range* as the filter for *frs*.

```
frs.setFilter(range);
```

To do the actual filtering, you call the method next, which in the reference implementation calls the appropriate version of the Predicate method implement behind the scenes. [Add material on three evaluate methods.]

If the return value is true, the row will be visible; if the return value is false, it will not. Thus, the following code fragment iterates through *frs* and prints only the rows in which the store is in California.

```
while (frs.next()) {
    int storeId = frs.getInt("STORE_ID");
    String city = frs.getString("CITY");
    long cof = frs.getLong("COF_SALES");
    long merch = frs.getLong("MERCH_SALES");
    long total = frs.getLong("TOTAL_SALES");

    print(storeId + "        ")
    print(city + "        ")
    print(cof + "        ")
    print(merch + "        ")
    println(total );
}
```

If none of the rows satisfied the criteria in *range*, there would have been no visible rows and nothing would have been printed.

Working with Filters

This section walks you through defining a different filter, setting it, and unsetting it. It also shows you how to update a FilteredRowSet object and the conditions under which you can modify values and insert or delete a row.

You set multiple filters serially. The first time you call the method setFilter and pass it a Predicate object, you have applied the filtering criteria in that filter. You can call setFilter again, passing it a different Predicate object, and that filter will be applied. Even though only one filter is set at a time, the effect is that both filters apply cumulatively .

For example, the owner has gotten a list of the Coffee Break stores in California by setting *range* as the Predicate object for *frs*. Now he wants to compare the stores in two California cities, San Francisco (SF in the table COFFEE_HOUSES) and Los Angeles (LA in the table). The first thing to do is to write a Predicate implementation that filters for stores in either SF or LA.

```
public class CityFilter implements Predicate {

    private Object city1
    private Object city2;
    private Object colName;
    private int colNumber;

    public CityFilter(Object [] city1, Object [] city2, Object [] colName) {
        this.city1 = city1;
        this.city2 = city2;
        this.colName = colName;
    }

    public CityFilter(Object [] city1, Object [] city2, int [] colNumber) {
        this.city1 = city1;
        this.city2 = city2;
        this.colNumber = colNumber;
    }

    public boolean evaluate(RowSet rs) {
        CachedRowSet crs = (CachedRowSet)rs;
        boolean bool1 = true;
        boolean bool2 = false;

    for (int i = 0; i < colName.length; i++) {
        if ((rs.getObject(colName[i] == city1[i]) ||
                    (rs.getObject(colName[i] == city2[i]) {
            return bool1;
            } else {
                return bool2;
            }
        }
    . . .// implementations for the other two versions of evaluate
    }
```

The following code fragment sets the new filter and iterates through the rows in *frs*, printing out the rows where the CITY column contains either SF or LA. Note that *frs* currently contains only rows where the store is in California, so the criteria of the Predicate object *state* are still in effect when the filter is changed to another Predicate object. The code that follows sets the filter to the CityFilter object *city*. The CityFilter implementation uses arrays as parameters to the constructors to illustrate how that can be done.

```
    Object [] city1 = {"SF"};
    Object [] city2 = {"LA"};
    Object [] colName = {"CITY"};
    CityFilter city = new CityFilter(city1, city2, colName);

    frs.setFilter(city);

    while (frs.next()) {
        int storeId = frs.getInt("STORE_ID");
        String city = frs.getString("CITY");
        BigDecimal cof = frs.getBigDecimal("COF_SALES");
        BigDecimal merch = frs.getBigDecimal("MERCH_SALES");
        BigDecimal total = frs.getBigDecimal("TOTAL_SALES");

        System.out.print(storeId + "       ");
        System.out.print(city + "       ");
        System.out.print(cof + "       ");
        System.out.print(merch + "       ");
        System.out.println(total );
    }
```

The printout should contain a row for each store that is in San Francisco, California or Los Angeles, California. If there were a row in which the CITY column contained LA and the STORE_ID column contained 40003, it would not be included in the list because it had already been filtered out when the filter was set to *range*. (40003 is not in the range of 10000 to 10999.)

Updating a FilteredRowSet Object

You can make a change to a FilteredRowSet object but only if that change does not violate any of the filtering criteria currently in effect. For example, you can insert a new row or change one or more values in an existing row if the new value or values are within the filtering criteria.

Inserting or Updating a Row

Assume that two new Coffee Break coffee houses have just opened and the owner wants to add them to his list. If a row to be inserted does not meet the cumulative filtering criteria in effect, it will be blocked from being added.

The current state of *frs* is that the Filter1 object *state* was set and then the CityFilter object *city* was set. As a result, *frs* currently makes visible only those rows that satisfy the criteria for both filters. And, equally important, you cannot add a row to *frs* unless it satisfies the criteria for both filters. [??true??] The following code fragment attempts to insert two new rows into *frs*, one row in which the values in the STORE_ID and CITY columns both meet the criteria, and one row in which the value in STORE_ID does not pass the filter but the value in the CITY column does.

```
frs.moveToInsertRow();

frs.updateInt("STORE_ID", 10101);
frs.updateString("CITY", "SF");
frs.updateBigDecimal("COF_SALES", new BigDecimal( 0 ));
frs.updateBigDecimal("MERCH_SALES", new BigDecimal( 0 ));
frs.updateBigDecimal("TOTAL_SALES", new BigDecimal( 0 ));
frs.insertRow();

frs.updateInt("STORE_ID", 33101);
frs.updateString("CITY", "SF");
frs.updateBigDecimal("COF_SALES", new BigDecimal( 0 ));
frs.updateBigDecimal("MERCH_SALES", new BigDecimal( 0 ));
frs.updateBigDecimal("TOTAL_SALES", new BigDecimal( 0 ));
frs.insertRow();

frs.moveToCurrentRow();
```

If you were to iterate through *frs* using the method next, you would find a row for the new coffee house in San Francisco, California, but not for the store in San Francisco, Washington.

The owner can add the store in Washington by nullifying the filter. With no filter set, all rows in *frs* are once more visible, and a store in any location can be added to the list of stores. The following line of code unsets the current filter, effectively nullifying both of the Predicate implementations previously set on *frs*.

```
frs.setFilter(null);
```

Deleting a Row

If the owner decides to close down or sell one of the Coffee Break coffee houses, he will want to delete it from the COFFEE_HOUSES table. He can delete the row for the underperforming coffee house as long as it is visible.

Given that the method setFilter has just been called with the argument null, there is no filter set on *frs*. This means that all rows are visible and can therefore be deleted. However, after the Filter1 object *state*, which filtered out any state other than California, was set, only stores located in California could be deleted. When the CityFilter object *city* was set for *frs*, only coffee houses in San Francisco, California or Los Angeles, California could be deleted because they were in the only rows visible.

Combining Two Filters into One

It is possible to apply different criteria to different columns in a FilteredRowSet object using the same Predicate object. For example, you can combine the criteria used in *state* and *city* into one Predicate object by initializing the constructor with arrays. That way, the first element in each array is applied and then the second element in each array is applied. The following example shows a possibility for implementing the Predicate interface to use two different criteria for filtering out rows.

```
public class Range3 implements Predicate {
    private Object lo[];
    private Object hi[];
    private int idx[];

    public Range3(Object[] lo, Object[] hi, int[] idx) {
        this.lo = lo;
        this.hi = hi;
        this.idx = idx;
    }
    public boolean evaluate(RowSet rs) {

        boolean bool1 = false;
        boolean bool2 = false ;

        try { CachedRowSet crs = (CachedRowSet)rs;

// Check the present row todetermine if it lies
// within the filtering criteria.

            for (int i = 0; i < idx.length; i++) {
                if ( ((rs.getObject(idx[i]).toString()).compareTo(lo[i].toString()) < 0) ||
                    ((rs.getObject(idx[i]).toString()).compareTo(hi[i].toString()) > 0) ) {
                    bool2 = true; // outside filter constraints
                } else {
                    bool1 = true; // within filter constraints
                }
            }

        } catch( SQLException e) {

        }

        if (bool2) {
            return false;
        } else {
            return true;
        }
    }
// implementation for two other versions of evaluate not shown
}
```

Code Samples

This section has three code samples to illustrate using three different Predicate objects as filters for a FilteredRowSet object. In each case, the code for the Predicate object follows the code for the FilteredRowSet object. As with all the code samples, you need to substitute your own JDBC URL, user name, password, and Driver class name.

Code Sample 1

The first code sample shows how to filter out rows in which the value in the designated column does not fall within the specified range. This is a the basic case for a FilteredRowSet object. The Predicate object used as the filter for FilteredRowSetSample1 is Range1, which follows the code sample.

You will see the following coding features in this sample code.

- The code for creating and inserting values into the table COFFEE_HOUSES uses the method addBatch to add SQL statements to the list of statements that the Statement object maintains. All of the statements in the list are sent to the database as a single unit (a batch) when the method executeBatch is called. Sending all of the statements as a batch is more efficient that sending each statement to the database separately.
- The method Connection.setAutoCommit is called to turn off auto-commit mode. With auto-commit mode off, none of the statements is executed until the method executeBatch is called. Then the method con.commit is called to commit the batch as one transaction. This is another efficiency strategy. Executing all of the statements in one transaction is more efficient that executing each statement separately.
- Because *con* is still an open Connection object, it is passed to the method execute. The reader will use this connection instead of having to obtain a new one, as is the case when no connection is passed to the method execute.
- The method close is called on the connection. This is necessary because the reader does not close the connection if it is passed one. The reader closes a connection only if it created the connection.
- The FilteredRowSet object is closed. It is good coding practice to close the objects you have created.

Note that some drivers may not support sending statements to the database as a batch. You can check by calling the DatabaseMetaData method supportsBatchUpdates, which returns true if the driver supports batch updates. If your driver returns false, you will need to rewrite the code to send each SQL statement as a separate Statement object.

```
stmt.executeUpdate("insert into COFFEE_HOUSES values(.....)");
```

You can still have all of the statements executed as one transaction with auto-commit mode set to false. Instead of calling the method executeBatch, you will need to call the method con.commit to execute the Statement objects.

```
==========================================================

import java.sql.*;
import javax.sql.rowset.*;
import com.sun.rowset.*;
import java.math.BigDecimal;

public class FilteredRowSetSample1 {
    public static void main(String [] args) {

        Connection con;
        String strUrl = "jdbc:datadirect:oracle://" +
                    "129.158.229.21:1521;SID=ORCL9";
        String strUserId = "scott";
        tring strPassword = "tiger";
        String className = "com.ddtek.jdbc.oracle.OracleDriver";
        BigDecimal b1;
        BigDecimal b2;
        BigDecimal b3;

        try {
        // Load the class of the driver
            Class.forName(className);
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
            con = DriverManager.getConnection(
                        strUrl, strUserId, strPassword);
            con.setAutoCommit(false);
            Statement stmt = con.createStatement();

            stmt.addBatch("drop table Coffee_Houses");

            stmt.addBatch("create table Coffee_Houses(" +
                "store_id int, city varchar(20), coffee " +
                "decimal(6,2), merch decimal(6,2), " +
                "total decimal(6,2))");

            b1 = new BigDecimal("3450.55");
            b2 = new BigDecimal("2005.21");
            b3 = new BigDecimal("5455.76");
            stmt.addBatch("insert into Coffee_Houses " +
                "values(10023, ’Mendocino’, "+b1+", "+b2+", "+b3+")");

            b1 = new BigDecimal("4699.39");
```

```
b2 = new BigDecimal("3109.03");
b3 = new BigDecimal("7808.42");
stmt.addBatch("insert into Coffee_Houses " +
    "values(33002, 'Seattle', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("5386.95");
b2 = new BigDecimal("2841.27");
b3 = new BigDecimal("8228.22");
stmt.addBatch("insert into Coffee_Houses " +
    "values(100040, 'SF', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("3147.12");
b2 = new BigDecimal("3579.52");
b3 = new BigDecimal("6726.64");
stmt.addBatch("insert into Coffee_Houses " +
    "values(32001,'Portland', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("2863.35");
b2 = new BigDecimal("1874.62");
b3 = new BigDecimal("4710.97");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10042,'SF', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("1987.77");
b2 = new BigDecimal("2341.21");
b3 = new BigDecimal("4328.98");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10024, 'Sacramento', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("2692.69");
b2 = new BigDecimal("1121.21");
b3 = new BigDecimal("8312.90");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10039,'Carmel', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("1533.48");
b2 = new BigDecimal("1007.02");
b3 = new BigDecimal("2450.50");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10041,'LA', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("2733.83");
b2 = new BigDecimal("1550.48");
b3 = new BigDecimal("4284.31");
stmt.addBatch("insert into Coffee_Houses " +
    "values(33002,'Olympia', "+b1+", "+b2+", "+b3+")");

stmt.executeBatch();
```

```
            con.commit();

// Now all the data has been inserted into the DB.
// Create a FilteredRowSet object, set the properties and
// populate it with this data.

            FilteredRowSet frs = new FilteredRowSetImpl();
            frs.setUsername(strUserId);
            frs.setPassword(strPassword);
            frs.setUrl(strUrl);
            frs.setCommand("select * from Coffee_Houses");
            frs.execute(con);
            con.close();

            Range1 stateFilter = new Range1(10000, 10999, 1);
            frs.setFilter(stateFilter);

            while(frs.next()) {
                System.out.println("Store id is: " + frs.getInt(1));
            }

// Now try to insert a row that does not satisfy the criteria.
// An SQLException will be thrown.

            try {
                frs.moveToInsertRow();
                frs.updateInt(1, 22999);
                frs.updateString(2, "LA");
                frs.updateBigDecimal(3, new BigDecimal("4455.01"));
                frs.updateBigDecimal(4, new BigDecimal("1579.33"));
                frs.updateBigDecimal(5, new BigDecimal("6034.34"));
                frs.insertRow();
                frs.moveToCurrentRow();
            } catch(SQLException sqle) {
                System.out.print("A row that does not satisfy ");
                System.out.println("the filter is being inserted");
                System.out.println("Message: " + sqle.getMessage());
            }

            frs.close();

        } catch(Exception e ) {
            System.err.print("Caught unexpected Exception: ");
            System.err.println(+ e.getMessage());
        }
    }
}
```

==

What follows is Range1, the implementation of the Predicate interface used in FilteredRowSetSample1. This implementation checks whether a given value is in the range from 10000 to 10999, which is the range for a STORE_ID number indicating that the store is in California.

```
import javax.sql.rowset.*;
import com.sun.rowset.*;
import java.util.*;
import java.lang.*;
import java.sql.*;
import javax.sql.RowSet;
import java.io.*;

public class Range1 implements Predicate, Serializable {

    private int idx;
    private int hi;
    private int lo;
    private String colName;

    public Range1(int lo, int hi, int idx) {
      this.hi = hi;
      this.lo = lo;
      this.idx = idx;
      colName = new String("");
    }

    public Range1(int lo, int hi, String colName, int idx) {
      this.lo = lo;
      this.hi = hi;
      this.colName = colName;
      this.idx = idx;
    }

    public boolean evaluate(RowSet rs) {
        int comp;
        int columnVal = 0;
        boolean bool = false;
        CachedRowSetImpl crs = (CachedRowSetImpl) rs;

        try {
            columnVal = crs.getInt(idx);
            if(columnVal <= hi && columnVal >= lo) {
                bool = true;
            } else {
                bool = false;
                }
            } catch(SQLException e) {
            }
            return bool;
        }

        public boolean evaluate(Object value, String columnName) {
            int colVal;
            boolean bool = false;
            if(columnName.equals(colName)) {
                colVal = (Integer.parseInt(value.toString()));

                if( colVal <= hi && colVal >= lo) {
                    bool = true;
                } else {
                    bool = false;
            } else {
                bool = true;
            }
            return bool;
        }

        public boolean evaluate(Object value, int columnIndex) {
            int colVal;
            boolean bool = false;
            if(columnIndex == idx) {
                colVal = (Integer.parseInt(value.toString()));

                if( colVal <= hi && colVal >= lo) {
                    bool = true;
                } else {
                    bool = false;
                }
            } else {
                bool = true;
            }
            return bool;
        }
    }
```

Code Sample 2

This code sample shows how to set one filter and then another filter to get the effect of both filters. The code creates the table COFFEE_HOUSES (just as FilteredRowSetSample1 did), sets a Range1 object as the first filter (just as FilteredRowSetSample1 did), and then sets a Range2 object as the second filter. Range2 filters for coffee houses in the city of San Francisco ("SF" in the table). Finally, the code prints the STORE_ID and CITY values for the rows visible in the Filtered-RowSet object *frs*.

==

```
import java.sql.*;
import javax.sql.rowset.*;
import com.sun.rowset.*;
import java.math.BigDecimal;

public class FilteredRowSetSample2 {
    public static void main(String [] args) {

        Connection con;
        String strUrl = "jdbc:datadirect:oracle://" +
                    "129.158.229.21:1521;SID=ORCL9";
        String strUserId = "scott";
        tring strPassword = "tiger";
        String className = "com.ddtek.jdbc.oracle.OracleDriver";
        BigDecimal b1;
        BigDecimal b2;
        BigDecimal b3;

        try {
        // Load the class of the driver
            Class.forName(className);
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
            con = DriverManager.getConnection(
                        strUrl, strUserId, strPassword);
            con.setAutoCommit(false);
            Statement stmt = con.createStatement();

            stmt.addBatch("drop table Coffee_Houses");

            stmt.addBatch("create table Coffee_Houses(" +
                "store_id int, city varchar(20), coffee " +
                "decimal(6,2), merch decimal(6,2), " +
                "total decimal(6,2))");

            b1 = new BigDecimal("3450.55");
            b2 = new BigDecimal("2005.21");
            b3 = new BigDecimal("5455.76");
```

```
stmt.addBatch("insert into Coffee_Houses " +
    "values(10023, 'Mendocino', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("4699.39");
b2 = new BigDecimal("3109.03");
b3 = new BigDecimal("7808.42");
stmt.addBatch("insert into Coffee_Houses " +
    "values(33002, 'Seattle', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("5386.95");
b2 = new BigDecimal("2841.27");
b3 = new BigDecimal("8228.22");
stmt.addBatch("insert into Coffee_Houses " +
    "values(100040, 'SF', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("3147.12");
b2 = new BigDecimal("3579.52");
b3 = new BigDecimal("6726.64");
stmt.addBatch("insert into Coffee_Houses " +
    "values(32001,'Portland', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("2863.35");
b2 = new BigDecimal("1874.62");
b3 = new BigDecimal("4710.97");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10042,'SF', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("1987.77");
b2 = new BigDecimal("2341.21");
b3 = new BigDecimal("4328.98");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10024, 'Sacramento', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("2692.69");
b2 = new BigDecimal("1121.21");
b3 = new BigDecimal("8312.90");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10039,'Carmel', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("1533.48");
b2 = new BigDecimal("1007.02");
b3 = new BigDecimal("2450.50");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10041,'LA', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("2733.83");
b2 = new BigDecimal("1550.48");
b3 = new BigDecimal("4284.31");
```

```
                stmt.addBatch("insert into Coffee_Houses " +
                     "values(33002,'Olympia', "+b1+", "+b2+", "+b3+")");

                stmt.executeBatch();
                con.commit();

// Now all the data has been inserted into the DB.
// Create a FilteredRowSet, set the properties and
// populate it with this data.

                FilteredRowSet frs = new FilteredRowSetImpl();
                frs.setUsername(strUserId);
                frs.setPassword(strPassword);
                frs.setUrl(strUrl);
                frs.setCommand("select * from Coffee_Houses");

                frs.execute(con);
                con.close();

// Now create the filter and set it. Range 1 is the
// class that implements the Predicate interface.

                Range1 stateFilter= new Range1(10000,10999,1);
                frs.setFilter(stateFilter);

// Set the second filter to filter out the coffee houses
// so that only those in the city of San Francisco
// are visible.

                 Range2 cityFilter = new Range2("SF", "SF", 2);
                 frs.setFilter(cityFilter);

          // Now only those stores whose store ID is between
          // 10000 and 10999 and whose city value is SF
          // are displayed
                while(frs.next()) {
                     System.out.println("Store ID is: " + frs.getInt(1));
                     System.out.print("City is: " );
                     System.out.println( frs.getString(2));
                }
                frs.close();

           } catch(Exception e ) {
                System.err.print("Caught unexpected Exception: ");
                System.err.println(+ e.getMessage());
           }
     }
}
```

The following code is an implementation of the Predicate interface that is used with FilteredRowSetSample2.

```
===========================================================

import javax.sql.rowset.*;
import com.sun.rowset.*;
import java.util.*;
import java.lang.*;
import java.sql.*;
import javax.sql.RowSet;
import java.io.Serializable;

public class Range2 implements Predicate, Serializable {
    private int idx;
    private Object hi;
    private Object lo;
    private String colName;

    public Range2(Object lo, Object hi, int idx) {
        this.hi = hi;
        this.lo = lo;
        this.idx = idx;
        this.colName = new String("");
    }

    public Range2(Object lo , Object hi , String colName, int idx) {
        this.lo = lo;
        this.hi = hi;
        this.colName = colName;
        this.idx = idx;
    }

    public boolean evaluate(RowSet rs) {
        int comp;
        String columnVal = "";
        boolean bool = false;
        FilteredRowSetImpl crs = (FilteredRowSetImpl) rs;
        try {
            columnVal = crs.getString(idx);

            System.out.println("Value is :"+columnVal);
            comp = columnVal.compareTo(lo);

        //System.out.println("comp1 :"+comp);
            if(comp < 0) {
                return false;
            }

            comp = columnVal.compareTo(hi);
        //System.out.println("comp2 :"+comp);
            if(comp > 0) {
                return false;
            }
        } catch(SQLException e) {
        } //end catch
         return true;
    }

    public boolean evaluate(Object value, String columnName) {
        int comp;
        if(!(columnName.equals(colName))) {
            return true;
        }
        comp = (value.toString()).compareTo(lo);
        if ( comp < 0 ) {
            return false;
        }
        comp = (value.toString()).compareTo(hi);
        if ( comp > 0 ) {
            return false;
        }

        return true;

    }

    public boolean evaluate(Object value, int columnIndex) {
        int comp;
        if(columnIndex != idx) {
            return true;
        }
        comp = (value.toString()).compareTo(lo);
        if( comp < 0 ) {
            return false;|
        }

        comp = (value.toString()).compareTo(hi);
        if ( comp > 0 ) {
            return false;
        }
        return true;
    }
}
```

Code Sample 3

FilteredRowSetSample3 uses a Predicate object that has two filtering criteria combined into one filter. In order to accommodate two sets of criteria, the arguments for the Predicate constructors are arrays.

```
=================================================================

import java.sql.*;
import javax.sql.rowset.*;
import com.sun.rowset.*;
import java.math.BigDecimal;

public class FilteredRowSetSample3 {
    public static void main(String [] args) {

        Connection con;
        String strUrl = "jdbc:datadirect:oracle://" +
                     "129.158.229.21:1521;SID=ORCL9";
        String strUserId = "scott";
        tring strPassword = "tiger";
        String className = "com.ddtek.jdbc.oracle.OracleDriver";
        BigDecimal b1;
        BigDecimal b2;
        BigDecimal b3;

        int [] idxArray = {1, 2};
        Object [] loArray = {new Integer(10000), "SF"};
        Object [] hiArray = {new Integer(10999), "SF"};

        try {
        // Load the class of the driver
        Class.forName(className);
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }

        try {
            con = DriverManager.getConnection(
                         strUrl, strUserId, strPassword);
            con.setAutoCommit(false);
            Statement stmt = con.createStatement();

            stmt.addBatch("drop table Coffee_Houses");

            stmt.addBatch("create table Coffee_Houses(" +
    "store_id int, city varchar(20), coffee " +
    "decimal(6,2), merch decimal(6,2), " +
    "total decimal(6,2))");

b1 = new BigDecimal("3450.55");
b2 = new BigDecimal("2005.21");
b3 = new BigDecimal("5455.76");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10023, 'Mendocino', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("4699.39");
b2 = new BigDecimal("3109.03");
b3 = new BigDecimal("7808.42");
stmt.addBatch("insert into Coffee_Houses " +
    "values(33002, 'Seattle', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("5386.95");
b2 = new BigDecimal("2841.27");
b3 = new BigDecimal("8228.22");
stmt.addBatch("insert into Coffee_Houses " +
    "values(100040, 'SF', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("3147.12");
b2 = new BigDecimal("3579.52");
b3 = new BigDecimal("6726.64");
stmt.addBatch("insert into Coffee_Houses " +
    "values(32001,'Portland', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("2863.35");
b2 = new BigDecimal("1874.62");
b3 = new BigDecimal("4710.97");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10042,'SF', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("1987.77");
b2 = new BigDecimal("2341.21");
b3 = new BigDecimal("4328.98");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10024, 'Sacramento', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("2692.69");
b2 = new BigDecimal("1121.21");
b3 = new BigDecimal("8312.90");
stmt.addBatch("insert into Coffee_Houses " +
    "values(10039,'Carmel', "+b1+", "+b2+", "+b3+")");

b1 = new BigDecimal("1533.48");
b2 = new BigDecimal("1007.02");
```

```
                b3 = new BigDecimal("2450.50");
                stmt.addBatch("insert into Coffee_Houses " +
                     "values(10041,'LA', "+b1+", "+b2+", "+b3+")");

                b1 = new BigDecimal("2733.83");
                b2 = new BigDecimal("1550.48");
                b3 = new BigDecimal("4284.31");
                stmt.addBatch("insert into Coffee_Houses " +
                     "values(33002,'Olympia', "+b1+", "+b2+", "+b3+")");

                stmt.executeBatch();
                con.commit();

// Now all the data has been inserted into the DB.
// Create a FilteredRowSet, set the properties and
// populate it with this data.

                FilteredRowSet frs = new FilteredRowSetImpl();
                frs.setUsername(strUserId);
                frs.setPassword(strPassword);
                frs.setUrl(strUrl);
                frs.setCommand("select * from Coffee_Houses");
                frs.execute(con);
                con.close();

                Range3 combinedFilter = new Range3(loArray,
                                             hiArray, idxArray);
                frs.setFilter(combinedFilter);

                while(frs.next()) {
                     System.out.println("Store ID is: " + frs.getInt(1));
                     System.out.println("City is: " + frs.getString(2));
                }
                frs.close();

           } catch(Exception e ) {
                System.err.print("Caught unexpected Exception: ");
                System.err.println(+ e.getMessage());
           }
     }
}
```

The following code defines Range3, a class implementing the interface Predicate. A Range3 object, which uses two criteria, allows only stores in San Francisco, California. A Range3 object is used as the filter for FilteredRowSetSample3.

```
===============================================================

import javax.sql.rowset.*;
import com.sun.rowset.*;
import java.util.*;
import java.lang.*;
import java.sql.*;
import javax.sql.RowSet;
import java.io.Serializable;

public class Range3 implements Predicate {
    private Object lo[];
    private Object hi[];
    private int idx[];

    public Range3(Object[] lo, Object[] hi, int[] idx) {
        this.lo = lo;
        this.hi = hi;
        this.idx = idx;
    }

    public boolean evaluate(RowSet rs) {

        boolean bool1 = false;
        boolean bool2 = false ;

        try { CachedRowSet crs = (CachedRowSet)rs;

// Check the present row determine if it lies
// within the filtering criteria.

            for (int i = 0; i < idx.length; i++) {
                if ( ((rs.getObject(idx[i]).toString()).compareTo(lo[i].toString()) < 0) ||
                    ((rs.getObject(idx[i]).toString()).compareTo(hi[i].toString()) > 0) ) {
                    bool2 = true; // outside filter constraints
                } else {
                    bool1 = true; // within filter constraints
                }
            }

        } catch( SQLException e) {

        }
```

```
            if (bool2) {
                return false;
            } else {
                return true;
            }
        }

    // No implementation needed.
        public boolean evaluate(Object value, String columnName) {
            return false;
        }

    // No implementation needed.
        public boolean evaluate(Object value, int columnIndex) {
            return false;
        }
    }
```

6 WebRowSet

A WebRowSet object is very special because in addition to offering all of the capabilities of a CachedRowSet object, it can write itself as an XML document and can also read that XML document to convert itself back to a WebRowSet object. Because XML is the language through which disparate enterprises can communicate with each other, it has become the standard for Web Services communication. As a consequence, a WebRowSet object fills a real need by making it easy for Web Services to send and receive data from a database in the form of an XML document.

In this chapter, you will learn how to do the following:

- Create and populate a WebRowSet object
- Write a WebRowSet object to an XML document
- Read data, properties, and metadata into a WebRowSet object from an XML document
- Make updates to a WebRowSet object
- Synchronize data back to the data source

Optional:

- Understand the structure and elements of the WebRowSet XML Schema

In the *Java Web Services Tutorial*, which uses the same Coffee Break scenario we have been using, the company has expanded to selling coffee online. Users order coffee by the pound from the Coffee Break web site. The price list is regularly updated by getting the latest information from the company's database. In the original tutorial, the price list was sent in a message that was an XML document composed using the SAAJ API. In this chapter you will see how much easier it is to send the price data using a WebRowSet object, which can write itself as an XML document with a single method call.

Creating and Populating a WebRowSet Object

You create a new WebRowSet object with the default constructor defined in the reference implementation, WebRowSetImpl, as shown in the following line of code.

```
WebRowSet priceList = new WebRowSetImpl();
```

Although *priceList* has no data yet, it has the default properties of a BaseRowSet object. Its SyncProvider object is at first set to the RIOptimisticProvider implementation, which is the default for all disconnected RowSet objects. However, the WebRowSet implementation resets the SyncProvider object to be the RIXMLProvider implementation. You will learn more about the RIXMLProvider implementation in the section “Synchronizing Back to the Data Source” [xref].

Getting back to our scenario, the Coffee Break headquarters regularly sends price list updates to its Web site. The Java Web Services Tutorial stated that this routine updating was done, but it did not show how it was done. This chapter on WebRowSet objects will show one way you can send the latest price list in an XML document.

The price list consists of the data in the columns COF_NAME and PRICE from the table COFFEES. The following code fragment sets the properties needed and populates *priceList* with the price list data.

```
priceList.setCommand("SELECT COF_NAME, PRICE FROM COFFEES");
priceList.setURL("jdbc:mySubprotocol:myDatabase");
priceList.setUsername("myUsername");
priceList.setPassword("myPassword");
priceList.execute();
```

At this point, in addition to the default properties, *priceList* contains the data in the columns COF_NAME and PRICE from the COFFEES table and also the metadata about these two columns.

Writing and Reading a WebRowSet Object to XML

Writing a WebRowSet object as an XML document is very easy: All you do is call the method writeXml. Similarly, all you do to read that XML document's contents into a WebRowSet object is to call the method readXml. Both of these methods do their work behind the scenes, meaning that everything except the results is invisible to you.

Using the writeXml Method

The method writeXML writes the WebRowSet object that invoked it as an XML document that represents its current state. It writes this XML document to the stream that you pass to it. The stream can be an OutputStream object, such as a FileOutputStream object, or a Writer object, such as a FileWriter object. If you pass the method writeXml an OutputStream object, you will write in bytes, which can handle all types of data; if you pass it a Writer object, you will write in characters. The following code demonstrates writing the WebRowSet object *priceList* as an XML document to the FileOutputStream object *oStream*.

```
java.io.FileOutputStream oStream = new java.io.FileOutputStream("priceList.xml");
priceList.writeXml(oStream);
```

The following code writes the XML document representing *priceList* to the FileWriter object *writer* instead of to an OutputStream object. The FileWriter class is a convenience class for writing characters to a file.

```
java.io.FileWriter writer = new java.io.FileWriter("priceList.xml");
priceList.writeXml(writer);
```

The other two versions of the method writeXml let you populate a WebRowSet object with the contents of a ResultSet object before writing it to a stream. In the following line of code, the method writeXml reads the contents of the ResultSet object *rs* into *priceList* and then writes *priceList* to the FileOutputStream object *oStream* as an XML document.

```
priceList.writeXml(rs, oStream);
```

In the next line of code, writeXml populates *priceList* with the contents of *rs*, but it writes the XML document to a FileWriter object instead of to an OutputStream object.

```
priceList.writeXml(rs, writer);
```

Using the readXml Method

The method readXml parses an XML document in order to construct the WebRowSet object the XML document describes. Analogous to the method writeXml, you can pass readXml an InputStream object or a Reader object from which to read the XML document.

```
java.io.FileInputStream iStream = new java.io.FileInputStream("priceList.xml");
priceList.readXml(iStream);

java.io.FileReader reader = new java.io.FileReader("priceList.xml");
priceList.readXml(reader);
```

Note that you can read the XML description into a new WebRowSet object or into the same WebRowSet object that called the writeXml method. In our secenario, where the price list information is being sent from headquarters to the web site, you would use a new WebRowSet object, as shown in the following lines of code.

```
WebRowSet recipient = new WebRowSetImpl();
java.io.FileReader reader = new java.io.FileReader("priceList.xml");
recipient.readXml(reader);
```

What Is in the XML Document

RowSet objects are more than just the data they contain. They have properties and metadata about their columns as well. Therefore, an XML document representing a WebRowSet object includes this other information in addition to its data. Further, the data in an XML document includes both current values and original values. (Recall from the CachedRowSet chapter that original values are the values that existed immediately before the most recent changes to data were made. These values are necessary for checking whether the corresponding value in the database has been changed, thus creating a conflict over which value should be persisted—the new value you put in the RowSet object or the new value someone else put in the database.)

The “WebRowSet XML Schema,” itself an XML document, defines what an XML document representing a WebRowSet object will contain and also the format in which it must be presented. Both the sender and the recipient use this schema because it tells the sender how to write the XML and the recipient how to parse the XML. Because the actual writing and reading is done internally by the implementations of the methods writeXml and readXml, you, as a user, do not need to understand what is in the “WebRowSet XML Schema” document. For reference, however, you can find the schema at the end of this chapter. If you want to learn more about XML, you can also refer to the XML chapter in the *Java Web Services Tutorial*.

Any XML document contains elements and subelements in a hierarchical structure. The following are the three main elements in an XML document describing a WebRowSet object:

- properties
- metadata
- data

Element tags signal the beginning and end of an element. For example, <properties> signals the beginning of the properties element, and </properties> signals its end. <map/> is a shorthand way of saying that the map subelement (one of the subelements in the properties element) has not been assigned a value. The XML shown in this chapter uses spacing and indentation to make it easier to read, but they are not used in an actual XML document, where spacing does not mean anything.

The next three sections show you what the three main elements contain for the WebRowSet *priceList*, created earlier in this chapter. Let’s assume that the data in *priceList* corresponds to the data in Table 6–1.

Table 6–1 PRICE_LIST

COF_NAME	**PRICE**
Colombian	7.99
French_Roast	8.99
Espresso	9.99
Colombian_Decaf	8.99

Table 6–1 PRICE_LIST

COF_NAME	PRICE
French_Roast_Decaf	9.99

Properties

Calling the method writeXml on *priceList* would produce an XML document describing *priceList*. The properties section of this XML document would look like the following.

```
<properties>
    <command>select COF_NAME, PRICE from COFFEES</command>
    <concurrency>1</concurrency>
    <datasource/>
    <escape-processing>true</escape-processing>
    <fetch-direction>0</fetch-direction>
    <fetch-size>0</fetch-size>
    <isolation-level>1</isolation-level>
    <key-columns/>
    <map/>
    <max-field-size>0</max-field-size>
    <max-rows>0</max-rows>
    <query-timeout>0</query-timeout>
    <read-only>false</read-only>
    <rowset-typ>TRANSACTION_READ_UNCOMMITTED</rowset-type>
    <show-deleted>false</show-deleted>
    <table-name/>
    <url>jdbc:thin:oracle</url>
    <sync-provider>
        <sync-provider-name>com.rowset.provider.RIOptimisticProvider
                                                </sync-provider-name>
        <sync-provider-vendor>Sun Microsystems</sync-provider-vendor>
        <sync-provider-version>1.0</sync-provider-version>
        <sync-provider-grade>LOW</sync-provider-grade>
        <data-source-lock>NONE</data-source-lock>
    <sync-provider/>
</properties>
```

You will notice that some properties have no value. For example, the datasource property is indicated with <datasource/>, which is a shorthand way of saying <data-source></datasource>. No value is given because the url property is set. Any connections that are established will be done using this JDBC URL, so no DataSource object needs to be set. Also, the username and password properties are not listed because they need to remain secret.

Metadata

The metadata section of the XML document describing a WebRowSet object contains information about the columns in that WebRowSet object. The following shows what this section looks like for the WebRowSet object *priceList*. Because *priceList* has two columns, the XML document describing it has two <column-definition> elements. Each <column-definition> element has subelements giving information about the column being described.

```
<metadata>
    <column-count>2</column-count>
    <column-definition>
        <column-index>1</column-index>
        <auto-increment>false&</auto-increment>
        <case-sensitive>true</case-sensitive>
        <currency>false</currency>
        <nullable>1</nullable>
        <signed>false</signed>
        <searchable>true</searchable>
        <column-display-size>10</column-display-size>
        <column-label>COF_NAME</column-label>
        <column-name>COF_NAME</column-name>
        <schema-name/>
        <column-precision>10</column-precision>
        <column-scale>0</column-scale>
        <table-name/>
        <catalog-name/>
        <column-type>1</column-type>
        <column-type-name>VARCHAR</column-type-name>
    </column-definition>


    <column-definition>
        <column-index>2</column-index>
        <auto-increment>false</auto-increment>
        <case-sensitive>true</case-sensitive>
        <currency>true</currency>
        <nullable>1</nullable>
        <signed>false</signed>
        <searchable>true</searchable>
        <column-display-size>10</column-display-size>
        <column-label>PRICE</column-label>
```

```
            <column-name>PRICE</column-name>
            <schema-name/>
            <column-precision>10</column-precision>
            <column-scale>2</column-scale>
            <table-name/>
            <catalog-name/>
            <column-type>3</column-type>
            <column-type-name>DECIMAL</column-type-name>
        </column-definition>
    </metadata>
```

From this metadata section, you can see that there are two columns in each row. The first column is COF_NAME, which holds values of type VARCHAR. The second column is PRICE, which holds values of type DECIMAL, and so on. Note that the column types in the schema are the data types used in the database, not types in the Java programming language ("Java types"). To get, set, or update values, though, you use getter, setter, and updater methods that use a Java type. For example, to set the value in the column COF_NAME, you use the method setString, and the driver converts the value to VARCHAR before sending it to the database.

Data

The data section gives the values for each column in each row of a WebRowSet object. If you have populated *priceList* and not made any changes to it, the data element of the XML document will look like the following. In the next section you will see how the XML document changes when you modify the data in *priceList*.

For each row there is a <currentRow> element, and because *priceList* has two columns, each <currentRow> element contains two <columnValue> elements.

```
    <data>
        <currentRow>
            <columnValue>
                Colombian
            </columnValue>
        <columnValue>
            7.99
        </columnValue>
    </currentRow>

    <currentRow>
        <columnValue>
            French_Roast
        </columnValue>
        <columnValue>
            8.99
        </columnValue>
    </currentRow>

    <currentRow>
        <columnValue>
            Espresso
        </columnValue>
        <columnValue>
            9.99
        </columnValue>
    </currentRow>

    <currentRow>
        <columnValue>
            Colombian_Decaf
        </columnValue>
        <columnValue>
            8.99
        </columnValue>
    </currentRow>

    <currentRow>
        <columnValue>
            French_Roast_Decaf
        </columnValue>
        <columnValue>
            9.99
        </columnValue>
    </currentRow>
</data>
```

Making Changes to a WebRowSet Object

You make changes to a WebRowSet object the same way you do to a CachedRowSet object. Unlike a CachedRowSet object, however, a WebRowSet object keeps track of updates, insertions, and deletions so that the writeXml method can write both the current values and the original values. The three sections that follow demonstrate making changes to the data and show what the XML document describing the WebRowSet object looks like after each change. You do not need to do anything at all regarding the XML document; any change to it is made automatically behind the scenes, just as with writing and reading the XML document.

Inserting a Row

If the owner of the Coffee Break chain wants to add a new coffee to his price list, the code might look like this.

```
priceList.moveToInsertRow();
priceList.updateString("COF_NAME", "Kona");
priceList.updateBigDecimal("PRICE", new BigDecimal("8.99");
priceList.insertRow();
priceList.moveToCurrentRow();
priceList.acceptChanges();
```

To reflect the insertion of the new row, the XML document will have the following <insertRow> element added to it.

```
<insertRow>
    <columnValue>
        Kona
    </columnValue>
    <columnValue>
        8.99
    </columnValue>
</insertRow>
```

Where a row is inserted in the database depends on the database.

Deleting a Row

The owner decides that Espresso is not selling enough and should be dropped from the coffees sold at the Coffee Break. He therefore wants to delete Espresso from the price list. Espresso is in the third row of *priceList*, so the following lines of code delete it.

```
priceList.absolute(3);
priceList.deleteRow();
```

The following <deleteRow> element will appear after the second row in the data section of the XML document, indicating that the third row has been deleted.

```
<deleteRow>
    <columnValue>
        Espresso
    </columnValue>
    <columnValue>
        9.99
    </columnValue>
</deleteRow>
```

Modifying a Row

The owner further decides that the price of Colombian coffee is too expensive and wants to lower it to 6.99 a pound. The following code sets the new price for Colombian coffee, which is in the first row, to 6.99 a pound.

```
priceList.first();
priceList.updateBigDecimal("PRICE", new BigDecimal("6.99"));
```

The XML document will reflect this change in a <modifyRow> element that gives both the old value (in the <columnValue> element) and the new value (in the <updat-eValue> element), as shown here. The value for the first column did not change, so there is an <updateValue> element for only the second column.

```
<modifyRow>
    <columnValue>
        Colombian
    </columnValue>
    <columnValue>
        7.99
    </columnValue>
    <updateValue>
        6.99
    </updateValue>
</modifyRow>
```

At this point, with the insertion of a row, the deletion of a row, and the modification of a row, the XML document for *priceList* would look like the following.

--->fill in what the data section reflecting these changes would look like

WebRowSet Code Example

The following code sample shows how a WebRowSet object can be used. The code creates a WebRowSet object, sets its properties, populates it, and modifies its data. Then it does what only a WebRowSet object can do: It serializes the WebRowSet object into an XML file (by calling the method writeXml). Then it populates another WebRowSet object with the first WebRowSet object (by calling the method readXML). Finally, the code compares the size of the two WebRowSet objects to see if they are the same. The result of calling the method readXml on the XML document created by writeXml is a WebRowSet object with the same data, metadata, and properties as those with which you started.

===

```
import java.sql.*;
import java.io.*;
import java.math.BigDecimal;
import javax.sql.rowset.*;
import com.sun.rowset.*;

public class WebRowSetSample {

    public static void main(String [] args) {
        String strUrl = "jdbc:datadirect:oracle://" +
                    "129.158.229.21:1521;SID=ORCL9";
        String strUserId = "scott";
        String strPassword = "tiger";
        String className = "com.ddtek.jdbc.oracle.OracleDriver";
        BigDecimal b;
        int [] keyCols = {1};
        FileReader fReader;
        FileWriter fWriter;
        try {
            Class.forName(className);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
        try {
            Connection con = DriverManager.getConnection(
                                strUrl, strUserId, strPassword);
            con.setAutoCommit(false);

            Statement stmt = con.createStatement();
            stmt.addBatch("drop table priceList");
            stmt.addBatch("create table priceList(" +
                "cof_name varchar(30), price decimal(6,2))");

            b = new BigDecimal("7.99");
            stmt.addBatch("insert into priceList values(" +
                " Colombian , "+b+")");

            b = new BigDecimal("8.99");
            stmt.addBatch("insert into priceList values(" +
                " French_Roast , "+b+")");

            b = new BigDecimal("9.99");
            stmt.addBatch("insert into priceList values(" +
                " Espresso , "+b+")");

            b = new BigDecimal("8.99");
            stmt.addBatch("insert into priceList values(" +
                " Colombian_Decaf , "+b+")");

            b = new BigDecimal("9.99");
            stmt.addBatch("insert into priceList values(" +
              " French_Roast_Decaf , "+b+")");

            stmt.executeBatch();
            con.commit();
            con.close();

// Create a WebRowSet and set its properties.
            WebRowSet sender = new WebRowSetImpl();
            sender.setUrl(strUrl);
            sender.setUsername(strUserId);
            sender.setPassword(strPassword);
            sender.setCommand("select * from priceList");
            sender.setKeyColumns(keyCols);
```

```
// Populate the WebRowSet
            sender.execute();
            System.out.print("WebRowSet size is: ");
            System.out.println( sender.size());

//Delete the row with "Espresso"
            sender.beforeFirst();
            while(sender.next()) {
                if(sender.getString(1).equals("Espresso")) {
                    System.out.print("Deleting row with ");
                    System.out.println(" Espresso");
                    sender.deleteRow();
                    break;
                }
            }

// Update price of Colombian
            sender.beforeFirst();
            while(sender.next()) {
                if(sender.getString(1).equals("Colombian")) {
                    System.out.print("Updating row with ");
                    System.out.println("Colombian");
                    sender.updateBigDecimal(2,
                            new BigDecimal("6.99"));
                    sender.updateRow();
                    break;
                }
            }
            int size1 = sender.size();
            fWriter = new FileWriter("priceList.xml");
            sender.writeXml(fWriter);
            fWriter.flush();
            fWriter.close();

// Create the receiving WebRowSet object
            WebRowSet receiver = new WebRowSetImpl();
            receiver.setUrl(strUrl);
            receiver.setUsername(strUserId);
            receiver.setPassword(strPassword);

//Now read the XML file.
            fReader = new FileReader("priceList.xml");
            receiver.readXml(fReader);
            int size2 = receiver.size();
            if(size1 == size2) {
                System.out.print("WebRowSet serialized and ");
                System.out.println("deserialized properly");
                } else {
                    System.out.print("Error....serializing/");
                    System.out.println("deserializing the WebRowSet");
                }

            } catch(SQLException sqle) {
                System.err.print("SQL Exception: ");
                System.err.println(+ sqle.getMessage());
                sqle.printStackTrace();
            }
        }
    }
```

===

WebRowSet XML Schema

The following XML document, "WebRowSet XML Schema," determines what an XML document representing a WebRowSet object contains.

```
<?xml version= 1.0  encoding= UTF-8 ?>

<!-- WebRowSet XML Schema by Jonathan Bruce (Sun Microsystems Inc.) -->

<xs:schema targetNamespace= http://java.sun.com/xml/ns/jdbc  xmlns:wrs= http://
java.sun.com/xml/ns/jdbc  xmlns:xs= http://www.w3.org/2001/XMLSchema
elementFormDefault= qualified >

    <xs:element name= webRowSet >
        <xs:complexType>
            <xs:sequence>
                <xs:element ref= wrs:properties />
                <xs:element ref= wrs:metadata />
                <xs:element ref= wrs:data />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name= columnValue  type= xs:anyType />
    <xs:element name= updateValue  type= xs:anyType />

    <xs:element name= properties >
        <xs:complexType>
            <xs:sequence>
                <xs:element name= command  type= xs:string />
```

```
                    <xs:element name= concurrency  type= xs:string />
                    <xs:element name= datasource  type= xs:string />
                    <xs:element name= escape-processing  type= xs:string />
                    <xs:element name= fetch-direction  type= xs:string />
                    <xs:element name= fetch-size  type= xs:string />
                    <xs:element name= isolation-level  type= xs:string />
                    <xs:element name= key-columns >
                        <xs:complexType>
                            <xs:sequence minOccurs= 0  maxOccurs= unbounded >
                                <xs:element name= column  type= xs:string />
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name= map >
                        <xs:complexType>
                            <xs:sequence minOccurs= 0  maxOccurs= unbounded >
                                <xs:element name= type  type= xs:string />
                                <xs:element name= class  type= xs:string />
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                    <xs:element name= max-field-size  type= xs:string />
                    <xs:element name= max-rows  type= xs:string />
                    <xs:element name= query-timeout  type= xs:string />
                    <xs:element name= read-only  type= xs:string />
                    <xs:element name= rowset-type  type= xs:string />
                    <xs:element name= show-deleted  type= xs:string />
                    <xs:element name= table-name  type= xs:string />
                    <xs:element name= url  type= xs:string />
                    <xs:element name= sync-provider >
                        <xs:complexType>
                            <xs:sequence>
                                <xs:element name= sync-provider-name  type= xs:string />
                                <xs:element name= sync-provider-vendor  type= xs:string />
                                <xs:element name= sync-provider-version  type= xs:string />
                                <xs:element name= sync-provider-grade  type= xs:string />
                                <xs:element name= data-source-lock  type= xs:string />
                            </xs:sequence>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name= metadata >
            <xs:complexType>
                <xs:sequence>
                    <xs:element name= column-count  type= xs:string />
                    <xs:choice>
                <xs:element name= column-definition  minOccurs= 0
                  maxOccurs= unbounded >
                   <xs:complexType>
                      <xs:sequence>
                         <xs:element name= column-index  type= xs:string />
                         <xs:element name= auto-increment  type= xs:string />
                         <xs:element name= case-sensitive  type= xs:string />
                         <xs:element name= currency  type= xs:string />
                         <xs:element name= nullable  type= xs:string />
                         <xs:element name= signed  type= xs:string />
                         <xs:element name= searchable  type= xs:string />
                         <xs:element name= column-display-size  type= xs:string />
                         <xs:element name= column-label  type= xs:string />
                         <xs:element name= column-name  type= xs:string />
                         <xs:element name= schema-name  type= xs:string />
                         <xs:element name= column-precision  type= xs:string />
                         <xs:element name= column-scale  type= xs:string />
                         <xs:element name= table-name  type= xs:string />
                         <xs:element name= catalog-name  type= xs:string />
                         <xs:element name= column-type  type= xs:string />
                         <xs:element name= column-type-name  type= xs:string />
                      </xs:sequence>
                   </xs:complexType>
                </xs:element>
             </xs:choice>
          </xs:sequence>
       </xs:complexType>
    </xs:element>

    <xs:element name= data >
       <xs:complexType>
          <xs:sequence minOccurs= 0  maxOccurs= unbounded >
             <xs:element name= currentRow  minOccurs= 0  maxOccurs= unbounded >
                <xs:complexType>
                   <xs:sequence minOccurs= 0  maxOccurs= unbounded >
                      <xs:element ref= wrs:columnValue />
                   </xs:sequence>
                </xs:complexType>
             </xs:element>
             <xs:element name= insertRow  minOccurs= 0  maxOccurs= unbounded >
                <xs:complexType>
                   <xs:choice minOccurs= 0  maxOccurs= unbounded >
                      <xs:element ref= wrs:columnValue />
                      <xs:element ref= wrs:updateValue />
                   </xs:choice>
                </xs:complexType>
             </xs:element>
             <xs:element name= deleteRow  minOccurs= 0  maxOccurs= unbounded >
```

```
                    <xs:complexType>
                         <xs:sequence minOccurs= 0  maxOccurs= unbounded >
                              <xs:element ref= wrs:columnValue />
                              <xs:element ref= wrs:updateValue />
                         </xs:sequence>
                    </xs:complexType>
               </xs:element>
               <xs:element name= modifyRow  minOccurs= 0  maxOccurs= unbounded >
                    <xs:complexType>
                         <xs:sequence minOccurs= 0  maxOccurs= unbounded >
                              <xs:element ref= wrs:columnValue />
                              <xs:element ref= wrs:updateValue />
                         </xs:sequence>
                    </xs:complexType>
               </xs:element>
          </xs:sequence>
     </xs:complexType>
</xs:element>
</xs:schema>
```