Note: The JDBC 2.0 API includes many new features in the `java.sql` package as well as the new Standard Extension package, `javax.sql`. This new JDBC API moves Java applications into the world of heavy-duty database computing. New features in the `java.sql` package include support for SQL3 data types, scrollable result sets, programmatic updates, and batch updates. The new JDBC Standard Extension API, an integral part of Enterprise JavaBeans (EJB) technology, allows you to write distributed transactions that use connection pooling, and it also makes it possible to connect to virtually any tabular data source, including files and spread sheets.

Readers who are new to JDBC might want to refer to the JDBC Basics chapter in the online version of The Java Tutorial.

For further training on JDBC, take a look at Chapter 3 which focuses on this API's new features and how to set up distributed application connections.

Rowsets make it easy to send tabular data over a network. They can also be used to provide scrollable result sets or updatable result sets when the underlying JDBC driver does not support them. These are the main uses for a rowset, and this tutorial will walk you through examples of these uses. Because there can be so many variations in rowsets, the first part of this chapter gives a conceptual description of rowsets and their uses. The next part walks you through creating and using a rowset. The last part shows you how a rowset can be used in a distributed Enterprise JavaBeans (EJB) application.

### 5.1 Types and Uses of Rowsets

A `RowSet` object contains a set of rows from a result set or some other source of tabular data, like a file or spreadsheet. Because a `RowSet` object follows the JavaBeans model for properties and event notification, it is a JavaBeans component that can be combined with other components in an application. As is true with other Beans, application programmers will probably use a development tool to create a `RowSet` object and set its properties.

Rowsets may have many different implementations to fill different needs. These implementations fall into two broad categories, rowsets that are connected and those that are disconnected. A disconnected rowset gets a connection to a data source in order to fill itself with data or to propagate changes in data back to the data source, but most of the time it does not have a connection open. While it is disconnected, it does not need a JDBC driver or the full JDBC API, so its footprint is very small. Thus a rowset is an ideal format for sending data over a network to a thin client.

Because it is not continually connected to its data source, a disconnected rowset stores its data in memory. It needs to maintain metadata about the columns it contains and information about its internal state. It also needs a facility for making connections, for executing commands, and for reading and writing data to and from the data source. A connected rowset, by contrast, opens a connection and keeps it open for as long as the rowset is in use.

Although anyone can implement a rowset, most implementations will probably be provided by vendors offering `RowSet` classes designed for fairly specific purposes. To make writing an implementation easier, the Java Software division of Sun Microsystems, Inc., plans to provide reference implementations for three different styles of rowsets in the future. The following list of planned implementations gives you an idea of some of the possibilities.

1. A `CachedRowSet` class—a disconnected rowset that caches its data in memory; not suitable for very large data sets, but an ideal way to provide thin Java clients, such as a Personal Digital Assistant (PDA) or Network Computer (NC), with tabular data
2. A `JDBCRowSet` class—a connected rowset that serves mainly as a thin wrapper around a `ResultSet` object to make a JDBC driver look like a JavaBeans component
3. A `WebRowSet` class—a connected rowset that uses the HTTP protocol internally to talk to a Java servlet that provides data access; used to make it possible for thin web clients to retrieve and possibly update a set of rows

### 5.2 Using a Rowset

As the conceptual description of rowsets pointed out, what you can do with a rowset depends on how it has been implemented. It can also depend on which properties have been set. The example rowsets used in this chapter are based on the `CachedRowSet` implementation, but because they are used for different purposes, one has several properties set whereas the other has none. Among other things, this tutorial will show you which properties to use and when to use them.
As part of the implementation, the application programmer will do the following:

1. Create the CachedRowSet Bean and set its properties
2. Register the Projector Bean as a listener to the rowset
3. Create the GUI for the application and implement the actions of the open and close buttons
4. Specify the rowset as a property on the form Bean

To put this all together, the application programmer will probably use a visual Bean development tool, which means that he will use very little RowSet API directly. Of course, the owner will use the application without writing any RowSet code himself. The upshot of all of this is that generally tools will generate the RowSet code you see in this tutorial. Also, remember that the code shown here is for illustrative purposes only because it uses the CachedRowSet class, for which there is no implementation currently available. Although the JDBC Standard Extension specification gives a preliminary outline of its functionality, some details in its implementation may be different when it is completed.

### 5.2.1 Creating a Rowset and Setting Properties

Because a programmer will generally use a Bean visual development tool to create a RowSet object and set its properties, the example code fragments shown here would most likely be executed by a Bean development tool. The main purpose of this section is to show you when and why you would want to set certain properties.

The code for creating a CachedRowSet object simply uses the default constructor.

```java
CachedRowSet crset = new CachedRowSet();
```

Now the programmer can set the CachedRowSet object's properties to suit the owner's needs. The RowSet interface, which the CachedRowSet class implements, contains getSet methods for retrieving and setting properties. These properties and a RowSet object's event notification capabilities (explained in a later section) are what make a RowSet object a JavaBeans component. A rowset's properties include its command, concurrency, type, data source name, url, user name, password, transaction isolation level, escape processing, maximum field size, maximum rows, query timeout, and type map. You are required to set only those properties that are needed for your particular use of a rowset. The following example uses several properties and explains why they are needed.

The owner wants the convenience of being able to make updates by scrolling to the rows he wants to update, so the property for the type needs to be set to scrollable. It may be that the CachedRowSet class will be implemented so that it is by default TYPE_SCROLL_INSENSITIVE, in which case the programmer would not need to set the rowset's type property. It does no harm to set it, however. The default for the concurrency property is ResultSet.CONCUR_READ_ONLY, so it needs to be set to CONCUR_UPDATABLE for our entrepreneur's use. The following lines of code make the CachedRowSet object `crset` scrollable and updatable.

```java
crset.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
crset.setConcurrency(ResultSet.CONCUR_UPDATABLE);
```

The owner will want to make updates to the table COFFEES, so the programmer sets the rowset's command string with the query

```sql
SELECT * FROM COFFEES.
```

When the method `execute` is called, this command will be executed, and the rowset will be populated with the data in the table COFFEES. The owner can then use the rowset to make his updates. In order to execute its
command, the rowset will need to make a connection with the database COFFEEBREAK, so the programmer also needs to set the properties required for that. If the DriverManager were being used to make a connection, he would set the properties for a JDBC URL, a user name, and a password. However, he wants to use the preferred means of getting a connection, which is to use a DataSource object, so he will set the properties for the data source name, the owner's user name, and the owner's password. For security, the programmer may opt to get the user name and password from the owner interactively at run time, in which case he would not need to set them as properties. (If you need a refresher on using a DataSource object to make a connection, start reading at page 160 in the advanced tutorial.) Here is the code a tool would generate to set the command string, the data source name, and the password properties for the CachedRowSet object crset.

```java
  crset.setCommand("SELECT * FROM COFFEES");
  crset.setDataSourceName("jdbc/coffeesDB");
  crset.setUsername("juanvaldez");
  crset.setPassword("espresso");
```

Note that the String object set for the data source name is the logical name that the system administrator (or someone acting in that capacity) registered with a JNDI naming service as the logical name for the COFFEEBREAK database. A programmer just needs to get the logical name, in this case jdbc/coffeesDB, from the system administrator and use it to set the data source property. When the rowset makes a connection, it will use the information in its properties, so the programmer or tool will not need to do anything except execute the command string, which you will see later. Internally the rowset gives the JNDI naming service the string the programmer set for the data source name property. Because jdbc/coffeesDB was previously bound to a DataSource object representing the database COFFEEBREAK, the naming service will return a DataSource object that the rowset can use to get a connection to COFFEEBREAK.

The programmer sets one more property, the transaction isolation level, which determines the transaction isolation level given to the connection that the rowset establishes. The owner does not want to read any data that has not been committed, so the programmer chooses the level TRANSACTION_READ_COMMITTED. The following line of code sets the rowset’s property so that “dirty reads” will not be allowed.

```java
  crset.setTransactionIsolation(
    Connection.TRANSACTION_READ_COMMITTED);
```

The other properties are all optional for the owner, so the programmer does not set any others. For example, he does not need to set a type map because there are no custom mappings in the table COFFEES. If the owner has the programmer change the command string so that it gets data from a table that has user-defined types with custom mappings, then the type map property will need to be set.

### 5.2.2 Rowsets and Event Notification

Being a JavaBeans component, a RowSet object has the ability to participate in event notification. In the application we are considering, the Projector Bean needs to be notified when the rowset is updated, so it needs to be registered with the rowset as a listener. The developer who wrote the Projector Bean will already have implemented the three RowSetListener methods rowChanged, rowSetChanged, and cursorMoved. These methods specify what the listener will do when an event occurs on the CachedRowSet object crset. The Projector Bean does not care about where the cursor is, so its implementation of cursorMoved is probably to do nothing. When one or more values in a row changes, the method rowChanged will be called on the listener. This method will probably check to see if the value in the PRICE or SALES columns has changed, and if either has changed, it will plug the appropriate value(s) into its projection model to get a value to display. The method rowSetChanged is invoked when the contents of the whole rowset change, which happens only when the rowset's command string has been changed and then executed. This method will probably have an implementation similar to that of the method rowChanged.

The following line of code registers projector, the bar chart component, as a listener for crset.

```java
  crset.addRowSetListener(projector);
```

Now that projector is registered as a listener with the rowset, it will be notified every time an event occurs on crset.

The programmer will use his development tool to bring one more component into the application, the editor that will be used to update the rowset. He does this by simply specifying crset as a property on the editor. This tells the editor to which component it should send the changes keyed into it.

### 5.2.3 Obtaining a Scrollable and Updatable Rowset

So far the programmer has created a CachedRowSet Object and set its properties. Now all he has to do in order to get a scrollable and updatable rowset is to call the method execute on the rowset. As a result of this call, the rowset does all of the following behind the scenes:

- Gets a connection to the COFFEEBREAK database, using the properties for the data source name, the user name, and the password
- Executes the query SELECT * FROM COFFEES, which has been set as its command string property
- Fills the rowset with the data from the result set produced by the query

The invocation that accomplishes all of this is the following single line of code.

```java
  crset.execute();
```

This produces a CachedRowSet object that contains the same data as the ResultSet object generated by the query SELECT *
FROM COFFEES. In other words, they both contain the data in the table COFFEES. The difference is that because the application developer has set the properties on the rowset to make it scrollable and updatable, the owner can move the cursor to any position in the rowset and modify its data. This is true even if the ResultSet object from which the rowset got its data is not scrollable or updatable. In fact, it is especially when a JDBC driver does not support scrollable or updatable result sets that you might want to use a rowset as a means of getting those capabilities.

5.2.4 Using a Rowset for Scrolling and Updating

Scrolling in a rowset is exactly the same as scrolling in a result set. The cursor is positioned before the first row in a newly-populated rowset, and a call to the method next moves the cursor to the first row. Thus, to iterate through a rowset from first row to last row, you call the method next in a while loop, just as you would do for a ResultSet object. For example, the following code fragment iterates through the entire RowSet object crset, printing the name of every coffee in the table COFFEES.

```java
    crset.execute();
    while (crset.next()) {
        System.out.println(crset.getString("COF_NAME"));
    }
```

With a non-scrollable rowset or result set, you are limited to iterating through the data once and in a forward direction. With scrolling, you can move the cursor in any direction and can go to a row as many times as you like. If you want a review of how to move the cursor, see the advanced tutorial section "Moving the Cursor in Scrollable Result Sets" on page 109.

The owner of The Coffee Break wanted a scrolling rowset so that he could easily make updates to a particular row. The following section illustrates moving the cursor to update a row.

5.2.5 Updating a Rowset

Updating a CachedRowSet object is similar to updating a ResultSet object. The updateXXX methods and the methods insertRow and deleteRow are inherited from ResultSet and are used in the same way. For example, the owner has brought up the rowset, which contains the current data in the table COFFEES, on his laptop computer. He wants to change the price for French_Roast_Decaf, which is in the fifth row, so he moves the cursor there. The GUI tool displaying the rowset will generate the following line of code to move the cursor to the fifth row.

```java
    crset.absolut(5);
```

The Projector Bean will be notified that the cursor has moved but will do nothing about it.

The owner now moves the cursor to the price, which is the third column, and changes the column's value to 10.49. In response, the GUI tool generates the following update statement.

```java
    crset.updateFloat(3, 10.49f);
```

Next the owner clicks on the row DONE button to indicate that he is finished making updates to the current row. This causes the GUI tool to generate the following line of code.

```java
    crset.updateRow();
```

The method rowChanged is called on the Projector Bean to notify it that a row in the rowset has changed. The Projector Bean determines whether the price and/or number of pounds sold has changed and, if so, plugs the most current price and number of pounds sold into its projection calculations. After it arrives at new projected values for revenue from sales of the affected coffee, it updates the bar chart to reflect the new values.

Now the owner moves to the previous row, which is the fourth row, changes the price to 9.49 and the sales amount to 500, and clicks the row DONE button. Note that the fourth column in the rowset contains the number of pounds sold in the last week. The GUI tool generates the following code.

```java
    crset.previous(-1); // or crset.absolut(4); 
    crset.updateFloat(3, 9.49f); 
    crset.updateInt(4, 500); 
    crset.updateRow();
```

So far the owner has updated the fourth and fifth rows in the rowset, but he has not updated the values in the database. If this had been a ResultSet object, both the result set and the database would have been updated with the call to the method updateRow. However, because this is a disconnected rowset, the method CachedRowSet.acceptChanges has to be called for the database to be updated. The owner will click the UPDATE DATABASE button if he wants to propagate his changes back to the database. The GUI tool will generate the following line of code.

```java
    crset.acceptChanges();
```

The application is implemented so that the acceptChanges method is not actually invoked until the owner returns to work and copies the updated rowset to his office computer. On the office machine, the rowset can create a connection for writing updated values back to the database. In addition to updating the database with the new values in rows four and five of the rowset, the acceptChanges method will set the values that the rowset keeps as its "original" values. Original values are the values the rowset had just before the current set of updates.

Before writing new values to the database, the rowset's writer component works behind the scenes to compare the rowset's original values with those in the database. If no one has changed values in the table, the rowset's original values and the values in the database should be the same. If there is no conflict, that is, the rowset's original values and the database's values match,
the writer may choose to write the updated values to the database, depending on how it is implemented. The current values that the writer enters will be used as the original values when a new set of updates is made. For example, the price in the fifth row was 9.99 before it was updated to 10.49. The rowset's original price of 9.99 should match the price for **French_Roast_Decaf** coffee in the database. If it does, the writer can update the database price to 10.49 and change the rowset's original price to 10.49. The next time the price for **French_Roast_Decaf** is changed, the writer will compare the original value (10.49) with the current value in the database.

In this example scenario you have seen how a rowset can be used to pass a set of rows to a thin client, in this case a laptop computer. You have also seen how a rowset can provide scrolling and updatability, which the JDBC driver used at the owner's office does not support. The next part of this chapter will show you how a rowset might be used in an EJB application.

### 5.3 An EJB Example

For this example, we assume that you are familiar with the concepts discussed in "Basic Tutorial" and "Advanced Tutorial," especially the sections on using the JDBC Standard Extension API. This EJB example gives only a high-level explanation of the EJB classes and interfaces used in it; if you want a more thorough explanation, you should see the EJB specification available at the following URL:


Let's assume that the owner of The Coffee Break has set up an EJB application to make it easier for his managers to order coffee for their stores. The managers can bring up an order form that has two buttons: one button for viewing a table with the coffees and prices currently available and another button for placing an order.

The developer who designed the form used JavaBeans components to create the buttons, layout, and other components of the form. The application developer will use Enterprise JavaBeans components to make the buttons do their work. The EJB component (enterprise bean) will be deployed in a container provided by an EJB server. The container manages the life cycle of its enterprise Beans and also manages the boundaries of the transactions in which they participate. The EJB implementation also includes a `DataSource` class that works with an `XADataSource` class to provide distributed transactions.

#### 5.3.1 A Distributed Application

An EJB application is always a distributed application, an application that distributes its work among different machines. An EJB application uses the three-tier model. The first tier is the client, which is typically a web browser. In our example, the client is a form running on The Coffee Break's intranet. The second tier, or middle tier, is made up of the EJB server and the JDBC driver. The third tier is one or more database servers.

The method `getCoffees` is one of the three methods that is implemented by our enterprise bean, the class `CoffeesBean`. Let's look at the implementation of this method, which creates and populates a rowset, and then look at how its invocation and execution are spread out over the three tiers. This method will be explained in more detail later in this chapter.

```java
public RowSet getCoffees() throws SQLException {
    Connection con = null;
    try {
        con = ds.getConnection("managerID", "mgrPassword");
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
        CachedRowSet rset = new CachedRowSet();
        rset.populate(rs);
        rs.close();
        stmt.close();
        return rset;
    }
    finally {
        if (con != null) con.close();
    }
    return null;
}
```

1. **Client**—invokes the method `getCoffees`
   The manager clicks the button **Display Coffee Prices** on The Coffee Break order form.

2. **Middle tier**—starts executing the method `getCoffees`
   The EJB server creates a connection to the DBMS server and sends the query to it.

3. **DBMS server(s)**—executes the query sent from the middle tier
   The DBMS server executes the query and returns a result set with coffee names and their prices to the middle tier server.
4. Middle tier—finishes executing the method `getCoffees`
   The middle tier server creates a rowset, populates it with the data from the result set returned by the DBMS, and
   returns the rowset to the client.

5. Client—the order form displays the rowset received from the middle tier

Note that a distributed application is not restricted to three tiers: It may have two tiers, a client and server. Also note that a
distributed application is different from a distributed transaction. A distributed transaction, often referred to as a global
transaction, is a transaction that involves two or more DBMS servers. A global transaction will always occur within the context of
a distributed application because by definition it requires at least a client and two servers.

5.3.2 Differences in Rowsets
As you have just seen, the rowset used in our EJB example is a `CachedRowSet` object that is created and populated on the
middle tier server. This disconnected rowset is then sent to a thin client. All of this is also true of the rowset in the laptop
example, but there are some differences between the two rowsets. The main difference is that the rowset used in the order form
for The Coffee Break is not updatable by the client; it is simply a list of coffees and their prices that the manager can look at.
Therefore, the rowset does not need its concurrency property set. In fact, the rowset used in the order form does not need any
properties at all. The method `getCoffees` gets a `DataSource` object and then uses it to get a connection, so the rowset
does not need to perform these tasks. This means that the rowset does not use a data source name, user name, or password
and thus does not need the properties for them to set. The order form rowset also needs no command property because the
`CachedRowSet` implementation executes the query to get coffee names and prices. Recall that by contrast, the rowset in the
laptop example created a connection, executed its command string, and populated itself with data by having its `execute`
method invoked. The only `CachedRowSet` method used in the EJB example is `populate`, which just reads data from the
`ResultSet` object passed to it and inserts the data into the rowset.

5.3.3 EJB and Distributed Transactions
In the EJB framework, one or more enterprise Beans can be deployed in a container, which manages the Beans. The container
controls the life cycle of a Bean, and it also controls the boundaries of distributed transactions.

Every enterprise Bean has a transaction attribute to tell the container how it should be managed with regard to distributed
transactions. The developer of the enterprise Bean in our example has given the Bean the transaction attribute `TX_REQUIRED`,
which means that the Bean's methods must be executed in the scope of a global transaction. If the component that invokes one
of the enterprise Bean's methods is already associated with a global transaction, the enterprise Bean method will be associated
with that transaction. If not, the container must start a new distributed transaction and execute the enterprise Bean method in
the scope of that transaction. When the method has completed, the container will commit the transaction.

The fact that the Bean's container manages the start and end of transactions has implications for the Bean's behavior. First, the
Bean should not call the methods `commit` or `rollback`. Second, the Bean should not change the default setting for a
connection's auto-commit mode. Because the `DataSource` object is implemented to work with distributed transactions, any
connection it produces has its auto-commit mode disabled. This prevents the connection from automatically committing a
transaction, which would get in the way of the container's management of the transaction. Thus, the Bean should leave the
connection's auto-commit mode disabled in addition to not calling the methods `commit` or `rollback`.

5.3.4 A Stateless `SessionBean` Object
The EJB component in this example is a `stateless SessionBean` object, which is the simplest kind of enterprise Bean. Being a
session Bean means that it is an extension of the client that creates it, typically reading and updating data in a database on
behalf of the client. A session Bean is created when a client begins its session and is closed when the client ends its session.
Being stateless means that the Bean does not need to retain any information it might get from a client from one method
invocation to the next. Therefore, any Bean instance can be used for any client.

For example, the enterprise Bean we will use has three methods. One creates a `CoffeesBean` object, another one retrieves a
table of coffees and prices, and a third places a manager's order. In general, because our enterprise Bean is a `SessionBean`
object, it is created when a manager opens The Coffee Break order form and is closed when he/she quits it. It is stateless
because it does not have to remember coffee prices or what the client ordered.

5.3.5 Overview of an EJB Application
An EJB application has four parts, which are described briefly in the following list. The enterprise Bean developer writes the first
three, and anyone, including the enterprise Bean developer, may supply the fourth. The sections following this one show the
code for each interface or class.

1. The remote interface—declares the methods that a client can call on the enterprise Bean

   In our example, the remote interface is the interface `Coffees`, which declares the methods `getCoffees` and
   `placeOrder`. The container generates an implementation of this interface that delegates to the class `CoffeesBean`
   `CoffeesBean`, supplied by the developer, actually defines what the methods do. It is the third item in this list.

   Instances of the interface `Coffees` are `EJBObjects`.

2. The home interface—declares the method that creates a `Coffees object`
In our example, the home interface is the interface CoffeesHome, which is registered with a JNDI naming service. It declares the method create and creates Coffees objects. The container implements this interface so that the method create delegates its work to the method ejbCreate, which is implemented by the CoffeesBean class.

3. The enterprise Bean class—the class that implements the methods used in the other parts of the application

Instances of this class are enterprise Beans. In our example this class is CoffeesBean, which implements the methods ejbCreate, getCoffees, and placeOrder.

4. The client code—the class on the client that invokes the enterprise Bean object

In our example, the client class is CoffeesClient. This class typically includes GUI components. For our example, if it were fully implemented, the CoffeesClient class would include buttons for invoking the methods getCoffees and placeOrder. It would also include a text editor for typing in the parameters for the method placeOrder. This class could have many different implementations, with or without GUI components, and it could be written by the enterprise Bean developer or anyone else.

Now let’s look at some sample code for an EJB application. Note that we kept this example very simple in order to concentrate on the basic concepts.

### 5.3.6 The Remote Interface

The interface Coffees declares the methods that managers of The Coffee Break coffee houses can invoke. In other words, this interface contains the methods that a remote client can invoke.

This interface, which extends EJBObject, declares the methods getCoffees and placeOrder. It imports four packages because it uses elements from each one. Both methods can throw a RemoteException as well as a SQLException because they use methods from the package java.rmi, the package for remote method invocation on Java objects. The following code defines the interface Coffees.

```java
import java.rmi.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;

public interface Coffees extends EJBObject {
    public RowSet getCoffees() throws RemoteException,
                                      SQLException;
    public void placeOrder(String cofName,
                             int quantity,
                             String MgrId)
                             throws RemoteException, SQLException;
}
```

### 5.3.7 The Home Interface

The home interface CoffeesHome is a factory for Coffees objects. It declares only the single method create, which creates Coffees objects, thus making CoffeesHome the simplest possible form of the home interface. The method create may throw a RemoteException, from the java.rmi package, or a CreateException, from the javax.ejb package.

```java
import java.rmi.*;
import javax.ejb.*;

public interface CoffeesHome extends javax.ejb.EJBHome {  
    public Coffees create() throws RemoteException,
                                     CreateException;
}
```

### 5.3.8 The Client Class

So far you have seen two interfaces with one thing in common: These interfaces contain the methods that will be called by the client class. The two methods in Coffees are called in response to button clicks from a manager. The client calls the CoffeesHome.create method to get a Coffees object it can use for invoking the methods defined on Coffees.

The first thing the CoffeesClient class does is to retrieve a CoffeesHome object that has been registered with a JNDI naming service. The CoffeesHome object has been bound to the logical name ejb/Coffees, so when ejb/Coffees is given to the method lookup, it returns a CoffeesHome object. Because the instance of CoffeesHome is returned as an RMI
PortableRemoteObject, it has to be narrowed to a CoffeesHome object before being assigned to the variable chome. The method CoffeesHome.create can then be called on chome to create the Coffees object coffees. Once the client has a Coffees object, it can call the methods Coffees.getCoffees and Coffees.placeOrder on it.

The methods invoked by a CoffeesClient object are implemented in the class CoffeesBean, which you will see next.

```java
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.*;

class CoffeesClient {
    public static void main(String[] args) {
        try {
            Context ctx = new InitialContext();
            Object obj = ctx.lookup("ejb/Coffees");
            CoffeesHome chome = (CoffeesHome)
                PortableRemoteObject.narrow(
                    obj, CoffeesHome.class);
            Coffees coffees = chome.create();
            RowSet rset = coffees.getCoffees();
// display the coffees for sale
// get user input from GUI
            coffees.placeOrder("Colombian", 3, "12345");
// repeat until user quits
        } catch (Exception e) {
            System.out.print(e.getClass().getName() + ":");
            System.out.println(e.getMessage());
        }
    }
}
```

5.3.9 The Enterprise Bean

The final part of our EJB component is the class CoffeesBean, which implements the methods that are declared in the interfaces Coffees and CoffeesHome and that are invoked in the class CoffeesClient. Note that it implements the SessionBean interface, but because it is a stateless SessionBean object, the implementations of the methods ejbRemove, ejbPassivate, and ejbActivate are empty. These methods apply to a SessionBean object with conversational state, but not to a stateless SessionBean object such as an instance of CoffeesBean. We will examine the code more closely after you have looked at it.

```java
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.ejb.*;
public class CoffeesBean implements SessionBean {
    public CoffeesBean () {} 
    public void ejbCreate() throws CreateException {
        try {
            ctx = new InitialContext();
            ds = (DataSource)ctx.lookup("jdbc/CoffeesDB");
        } catch (Exception e) {
            throw new CreateException();
        }
    }
    public RowSet getCoffees() throws SQLException {
        Connection con = null;
        try {
            con = ds.getConnection("managerID", "mgrPassword");
```
CoffeesBean can be divided into the following steps:

1. Defining the default constructor
   ```java
   public CoffeesBean() {}
   ```

2. Defining the method `ejbCreate`
   ```java
   public void ejbCreate() throws CreateException {
      try {
         ctx = new InitialContext();
         ds = (DataSource)ctx.lookup("jdbc/CoffeesDB");
      }
   }
   ```
The `Context` object `ctx` and the `DataSource` object `ds` are private fields originally set to `null`. This method retrieves an instance of the `DataSource` class that is associated with the logical name `jdbc/CoffeesDB` and assigns it to `ds`. The `DataSource` object `ds` can be used to create connections to the database `COFFEEBREAK`. This work is done once when the Bean is created to avoid doing it over and over each time the methods `getCoffees` and `placeOrder` are called.

3. Defining the method `getCoffees`. We will break this method down into smaller units for examination.

```java
public RowSet getCoffees() throws SQLException {
    Connection con = null;
    try {
        con = ds.getConnection("managerID", "mgrPassword");
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");

        CachedRowSet crset = new CachedRowSet();
        crset.populate(rs);

        rs.close();
        stmt.close();
        return crset;
    } catch (Exception e) {
        throw new CreateException();
    } finally {
        if (con != null) con.close();
    }
    return null;
}
```

As the signature indicates, this method returns a `RowSet` object. It uses the `DataSource` object that the method `ejbCreate` obtained from the JNDI naming service to create a connection to the database that `ds` represents. Supplying a user name and password to the method `DataSource.getConnection` produces the `Connection` object `con`. This is a connection to the database `COFFEEBREAK` because when the system administrator deployed the `DataSource` object used to make the connection, she gave it the properties for the `COFFEEBREAK` database.

The code then creates a `Statement` object and uses it to execute a query. The query produces a `ResultSet` object that has the name and price for every coffee in the table `COFFEES`. This is the data that the client has requested.

```java
rs.close();
stmt.close();
return rset;
}
```

The method `getCoffees` returns the newly-populated `RowSet` object if the connection is made and the rowset is successfully filled with data. Otherwise, `getCoffees` returns `null`. There are two points to be made about these lines of code. First, it contains a `finally` block that assures that even if there is an exception thrown, if the connection is not `null`, it will be closed and thereby recycled. Because the EJB server and JDBC driver being used implement connection pooling, a valid connection will automatically be put back into the pool of available connections when it is closed.

The second point is that the code does not enable the auto-commit mode, nor does it call the methods `commit` or `rollback`. The reason is that this enterprise Bean is operating within the scope of a distributed transaction, so the container will commit or rollback all transactions.

4. Defining the method `placeOrder`

```java
public void placeOrder(String cofName,
                      int quantity,
                      String MgrId)
                     throws SQLException {
    Connection con = null;
    try {
```
The method `placeOrder` gets values for the three input parameters and then sets them. After the manager clicks the `PLACE ORDER` button on the order form, he gets three blank spaces into which to type the coffee name, the number of pounds, and his `ManagerID`. For example, the manager might have typed "Colombian", 50, and "12345" in the blanks on his form. The server would get the following line of code:

```java
coffees.placeOrder("Colombian", 50, "12345");
```

The `placeOrders` method would produce the following code:

```java
pstmt.setString(1, "Colombian");
pstmt.setInt(2, 50);
pstmt.setString(3, "12345");
```

The following update statement would effectively be sent to the DBMS server to be executed.

```sql
INSERT INTO ORDERS VALUES ("Colombian", 50, "12345")
```

This would put a new row into the `ORDERS` table to record the manager's order.

As with the method `getCoffees`, the `placeOrders` method has a `finally` block to make sure that a valid connection is closed even if there is an exception thrown. This means that if the connection is valid, it will be returned to the connection pool to be reused.

The rest of the implementation deals with methods inherited from the `SessionBean` interface. The methods `ejbRemove`, `ejbPassivate`, and `ejbActivate` apply to `SessionBean` objects with state. Because `CoffeesBean` is a stateless `SessionBean` object, the implementations for these methods are empty.

Congratulations! You have finished the tutorials on the complete JDBC API. You can create tables, update tables, retrieve and process data from result sets, use prepared statements and stored procedures, and use transactions. You can also use the more advanced functionality, including SQL3 data types, batch updates, programmatic updates, custom mapping, making a connection with a `DataSource` object, connection pooling, distributed transactions, and rowsets. You have also seen how the JDBC API works with EJB technology and gotten a high-level summary of the four parts of an EJB application.

The reference chapters give more examples and more in-depth explanations of the features you have learned to use in these tutorials. Remember to take advantage of the glossary and the index as aids for getting information quickly.

**About the Author**

Maydene Fisher has extensive experience as a technical writer specializing in the documentation of object-oriented programming languages.