```
}

/**
   The getCustomerNames method returns an array
   of Strings containing all the customer names.
*/

public String[] getCustomerNames() throws SQLException
{
   // Create a connection to the database.
   conn = DriverManager.getConnection(DB_URL);

   // Create a Statement object for the query.
   Statement stmt =
      conn.createStatement(
              ResultSet.TYPE_SCROLL_SENSITIVE,
              ResultSet.CONCUR_READ_ONLY);

   // Execute the query.
   ResultSet resultSet =
      stmt.executeQuery("SELECT Name FROM Customer");

   // Get the number of rows
   resultSet.last();                  // Move last row
   int numRows = resultSet.getRow(); // Get row number
   resultSet.first();                 // Move to first row

   // Create an array for the customer names.
   String[] listData = new String[numRows];

   // Populate the array with customer names.
   for (int index = 0; index < numRows; index++)
   {
      // Store the customer name in the array.
      listData[index] = resultSet.getString(1);

      // Go to the next row in the result set.
      resultSet.next();
   }

   // Close the connection and statement objects.
   conn.close();
   stmt.close();

   // Return the listData array.
   return listData;
}

```

```
/**
   The getCustomerNum method returns a specific
   customer's number.
   @param name The specified customer's name.
*/

public String getCustomerNum(String name)
                              throws SQLException
{
   String customerNumber = "";

   // Create a connection to the database.
   conn = DriverManager.getConnection(DB_URL);

   // Create a Statement object for the query.
   Statement stmt = conn.createStatement();

   // Execute the query.
   ResultSet resultSet =
     stmt.executeQuery("SELECT CustomerNumber " +
                       "FROM Customer " +
                       "WHERE Name = '" + name + "'");

   if (resultSet.next())
      customerNumber = resultSet.getString(1);

   // Close the connection and statement objects.
   conn.close();
   stmt.close();

   // Return the customer number.
   return customerNumber;
}

/**
   The submitOrder method submits an order to
   the UnpaidOrder table in the CoffeeDB database.
   @param custNum The customer number.
   @param prodNum The product number.
   @param quantity The quantity ordered.
   @param price The price.
   @param orderDate The order date.
*/

public void submitOrder(String custNum, String prodNum,
                        int quantity, double price,
                        String orderDate) throws SQLException
{
```

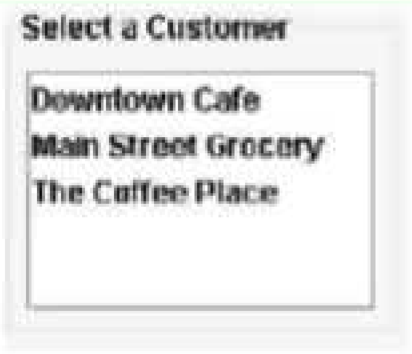
```
239         // Calculate the cost of the order.
240         double cost = quantity * price;
241
242         // Create a connection to the database.
243         conn = DriverManager.getConnection(DB_URL);
244
245         // Create a Statement object for the query.
246         Statement stmt = conn.createStatement();
247
248         // Execute the query.
249         stmt.executeUpdate("INSERT INTO UnpaidOrder VALUES('" +
250                             custNum + "', '" +
251                             prodNum + "', '" + orderDate + "', " +
252                             quantity + ", " + cost + ")");
253
254         // Close the connection and statement objects.
255         conn.close();
256         stmt.close();
257     }
258 }
```

Here is a summary of the methods in the `CoffeeDBManager` class:

- The constructor, in lines 21 through 25, establishes a connection to the database. The `getCoffeeNames` method, in lines 32 through 69, returns an array of strings containing the names of all the coffees in the `Coffee` table.
- The `getProdNum` method, in lines 77 through 106, accepts a `String` argument containing the name of a coffee. The method returns the coffee's product number.
- The `getCoffeePrice` method, in lines 114 through 143, accepts a `String` argument containing a coffee's product number. The method returns the price of the specified coffee.
- The `getCustomerNames` method, in lines 150 through 189, returns an array of strings containing the names of all the customers in the `Customer` table.
- The `getCustomerNum` method, in lines 197 through 223, accepts a `String` argument containing a customer's name. The method returns that customer's customer number.
- The `submitOrder` method in lines 235 through 257 creates a row in the `UnpaidOrder` table. It accepts arguments for the customer number, the product number of the coffee being ordered, the quantity being ordered, the coffee's price per pound, and the order date. Line 240 calculates the cost of the order by multiplying the quantity by the price per pound. Line 243 opens a connection to the database and line 246 creates a `Statement` object. Lines 249 through 252 execute an `INSERT` statement on the `UnpaidOrders` table.

The next class we will look at is the `CustomerPanel` class, shown in Code Listing 16-17. This class, which inherits from `JPanel`, uses a `JList` component to display all of the customer names in the `Customer` table. Figure 16-21 shows an example of how the panel will appear when it is displayed in a GUI application.

Figure 16-21 Customer panel



Code Listing 16-17 (CustomerPanel.java)

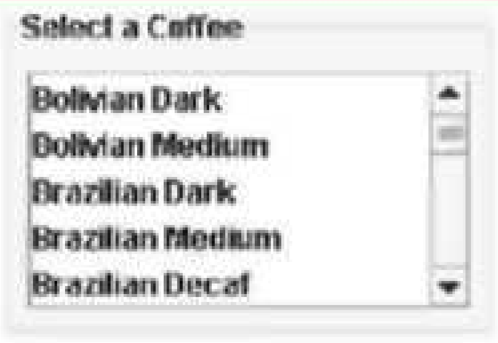
```
import java.sql.*;
import javax.swing.*;

/**
   The CustomerPanel class is a custom JPanel that
   shows a list of customers in a JList.
*/

public class CustomerPanel extends JPanel
{
   private final int NUM_ROWS = 5; // Number of rows to display
   private JList customerList;     // A list for customer names
   String[] names;                 // To hold customer names

   /**
      Constructor
   */

   public CustomerPanel()
   {
      try
      {
         // Create a CoffeeDBManager object.
         CoffeeDBManager dbManager = new CoffeeDBManager();

         // Get a list of customer names as a String array.
         names = dbManager.getCustomerNames();

         // Create a JList object to hold customer names.
         customerList = new JList(names);

         // Set the number of visible rows.
         customerList.setVisibleRowCount(NUM_ROWS);
34
35          // Put the JList object in a scroll pane.
36          JScrollPane scrollPane =
37                         new JScrollPane(customerList);
38
39          // Add the scroll pane to the panel.
40          add(scrollPane);
41
42          // Add a titled border to the panel.
43          setBorder(BorderFactory.createTitledBorder(
44                                    "Select a Customer"));
45       }
46       catch (SQLException ex)
47       {
48          ex.printStackTrace();
49          System.exit(0);
50       }
51    }
52
53    /**
54       The getCustomer method returns the customer
55       name selected by the user.
56    */
57
58    public String getCustomer()
59    {
60       // The JList class's getSelectedValue method returns
61       // an Object reference, so we will cast it to a String.
62       return (String) customerList.getSelectedValue();
63    }
64 }
```

Line 24 in the constructor creates an instance of the `CoffeeDBManager` class. Line 27 calls the `getCustomerNames` method to get a `String` array containing the customer names. Line 30 creates a `JList` component, passing the array of customer names as an argument to the constructor. This will cause the `JList` component to be populated with the names of all the customers in the `Customer` table. Line 33 sets the number of visible rows for the `JList` component, and lines 36 and 37 put the `JList` in a scroll pane. Line 40 adds the scroll pane to the panel, and lines 43 and 44 create a titled border around the panel.

The `getCustomer` method, in lines 58 through 63, returns the customer name that is currently selected in the `JList` component.

The next class, `CoffeePanel`, is shown in Code Listing 16-18. This class, which inherits from `JPanel`, uses a `JList` component to display all of the coffee names in the `Description` column of the `Coffee` table. Figure 16-22 shows an example of how the panel will appear when it is displayed in a GUI application.

Figure 16-22 Coffee panel


Select a Coffee

Bolivian Dark
Bolivian Medium
Brazilian Dark
Brazilian Medium
Brazilian Decaf

Code Listing 16-18 (CoffeePanel.java)

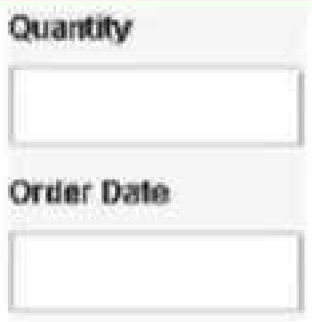
```
import java.sql.*;
import javax.swing.*;

/**
   The CoffeePanel class is a custom JPanel that
   shows a list of coffees in a JList.
*/

public class CoffeePanel extends JPanel
{
   private final int NUM_ROWS = 5; // Number of rows to display
   private JList coffeeList;       // A list for coffee descriptions
   String[] coffeeNames;           // To hold coffee names

   /**
      Constructor
   */

   public CoffeePanel()
   {
      try
      {
         // Create a CoffeeDBManager object.
         CoffeeDBManager dbManager = new CoffeeDBManager();

         // Get a list of coffee names as a String array.
         coffeeNames = dbManager.getCoffeeNames();

         // Create a JList object to hold the coffee names.
         coffeeList = new JList(coffeeNames);

         // Set the number of visible rows.
         coffeeList.setVisibleRowCount(NUM_ROWS);
34 
35          // Put the JList object in a scroll pane.
36          JScrollPane scrollPane = new JScrollPane(coffeeList);
37 
38          // Add the scroll pane to the panel.
39          add(scrollPane);
40 
41          // Add a titled border to the panel.
42          setBorder(BorderFactory.createTitledBorder(
43                                     "Select a Coffee"));
44       }
45       catch (SQLException ex)
46       {
47          ex.printStackTrace();
48          System.exit(0);
49       }
50    }
51 
52    /**
53       The getCoffee method returns the coffee
54       description selected by the user.
55    */
56 
57    public String getCoffee()
58    {
59       // The JList class's getSelectedValue method returns
60       // an Object reference, so we will cast it to a String.
61       return (String) coffeeList.getSelectedValue();
62    }
63 }
```

Line 24 in the constructor creates an instance of the `CoffeeDBManager` class. Line 27 calls the `getCoffeeNames` method to get a `String` array containing coffee names. Line 30 creates a `JList` component, passing the array of coffee names as an argument to the constructor. This will cause the `JList` component to be populated with the names of all the coffees in the `Coffee` table. Line 33 sets the number of visible rows for the `JList` component, and line 36 puts the `JList` in a scroll pane. Line 39 adds the scroll pane to the panel, and lines 42 and 43 create a titled border around the panel.

The `getCoffee` method, in lines 57 through 62, returns the coffee name that is currently selected in the `JList` component.

The next class, `QtyDatePanel`, is shown in Code Listing 16-19. This class, which inherits from `JPanel`, simply displays `JTextField` components for the quantity of coffee being ordered (in pounds) and the date of the order. Figure 16-23 shows an example of how the panel will appear when it is displayed in a GUI application.

Figure 16-23 QtyDate panel



Code Listing 16-19 `(QtyDatePanel.java)`

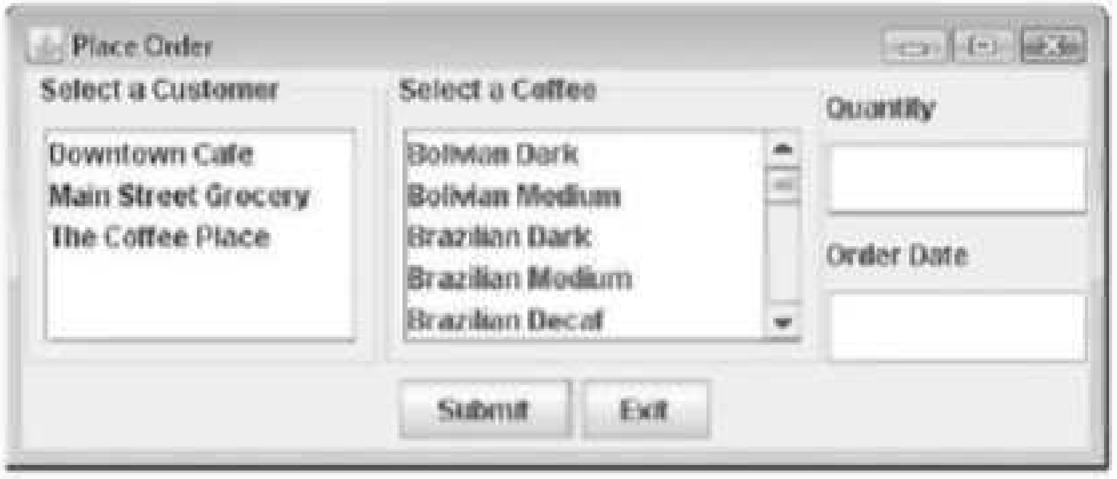
```java
import javax.swing.*;
import java.awt.*;

/**
   The QtyDatePanel presents text fields for the
   quantity of coffee being ordered and the order
   date.
*/

public class QtyDatePanel extends JPanel
{
   private JTextField qtyTextField;   // Order quantity
   private JTextField dateTextField;  // order date

   /**
      Constructor
   */

   public QtyDatePanel()
   {
      // Create a label prompting the user
      // for a quantity.
      JLabel qtyPrompt = new JLabel("Quantity");

      // Create a text field for the quantity.
      qtyTextField = new JTextField(10);

      // Create a label prompting the user
      // for a date.
      JLabel datePrompt = new JLabel("Order Date");

      // Create a text field for the date.
      dateTextField = new JTextField(10);

      // Create a grid layout manager, 4 rows, 1 column.
      setLayout(new GridLayout(4, 1));
37
38         // Add the components to the panel.
39         add (qtyPrompt);
40         add (qtyTextField);
41         add (datePrompt);
42         add (dateTextField);
43      }
44
45      /**
46         The getQuantity method returns the quantity
47         entered by the user.
48         @return The value entered into qtyTextField.
49      */
50
51      public int getQuantity()
52      {
53         return Integer.parseInt(qtyTextField.getText());
54      }
55
56      /**
57         The getDate method returns the quantity
58         entered by the user.
59         @return The value entered into dateTextField.
60      */
61
62      public String getDate()
63      {
64         return dateTextField.getText();
65      }
66
67      /**
68         The clear method clears the text fields.
69      */
70
71      public void clear()
72      {
73         qtyTextField.setText("");
74         dateTextField.setText("");
75      }
76   }
```

The constructor creates text fields into which the user can enter the quantity of an order and the order date. It also creates labels that prompt the user for the correct information for each text box. A `GridLayout` manager is then created, and these components are added to the panel.

The `getQuantity` method, in lines 51 through 54, returns the quantity entered by the user as an integer. The `getDate` method, in lines 62 through 65, returns the order date entered by the user as a `String`. The `clear` method, in lines 71 through 75, clears the text fields of any data.

Now let's look at the PlaceOrder class, shown in Code Listing 16-20. This application presents the GUI interface shown in Figure 16-24 for order entry.

Figure 16-24 Order Entry GUI



Code Listing 16-20 (PlaceOrder.java)

```
import java.sql.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
   The PlaceOrder class is a simple order entry system.
*/

public class PlaceOrder extends JFrame
{
   CustomerPanel customerPanel; // Panel for customers
   CoffeePanel coffeePanel;     // Panel for coffees
   QtyDatePanel qtyDatePanel;   // Panel for quantity and date
   JPanel buttonPanel;          // Panel for buttons

   /**
      Constructor
   */

   public PlaceOrder()
   {
      // Set the window title.
      setTitle("Place Order");

      // Specify an action for the close button.
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a CustomerPanel object.
      customerPanel = new CustomerPanel();

      // Create a CoffeePanel object.
      coffeePanel = new CoffeePanel();

      // Create a QtyDatePanel object.
      qtyDatePanel = new QtyDatePanel();

      // Build the ButtonPanel object.
      buildButtonPanel();

      // Create a BorderLayout manager.
      setLayout(new BorderLayout());

      // Add the panels to the content pane.
      add(customerPanel, BorderLayout.WEST);
      add(coffeePanel, BorderLayout.CENTER);
      add(qtyDatePanel, BorderLayout.EAST);
      add(buttonPanel, BorderLayout.SOUTH);

      // Pack and display the window.
      pack();
      setVisible(true);
   }

   /**
      The buildButtonPanel method builds a panel for
      the Submit and Exit buttons.
   */

   public void buildButtonPanel()
   {
      // Create a panel for the buttons.
      buttonPanel = new JPanel();

      // Create a Submit button and add an action listener.
      JButton submitButton = new JButton("Submit");
      submitButton.addActionListener(new SubmitButtonListener());

      // Create an Exit button.
      JButton exitButton = new JButton("Exit");
      exitButton.addActionListener(new ExitButtonListener());

      // Add the buttons to the panel.
      buttonPanel.add(submitButton);
      buttonPanel.add(exitButton);
   }

   /**
      Private inner class that handles submit button events
   */

   private class SubmitButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         try
         {
            // Get the customer name from the CustomerPanel object.
            String customerName = customerPanel.getCustomer();

            // Get the coffee description from the CoffeePanel.
            String coffee = coffeePanel.getCoffee();

            // Get the quantity from the QtyDatePanel object.
            int qty = qtyDatePanel.getQuantity();

            // Get the order date from the QtyDatePanel object.
            String orderDate = qtyDatePanel.getDate();

            // Create a CoffeeDBManager object.
            CoffeeDBManager dbManager = new CoffeeDBManager();

            // Get the customer number.
            String customerNum =
                  dbManager.getCustomerNum(customerName);

            // Get the coffee product number.
            String prodNum = dbManager.getProdNum(coffee);

            // Get the coffee price per pound.
            double price = dbManager.getCoffeePrice(prodNum);

            // Submit the order.
            dbManager.submitOrder(customerNum, prodNum, qty,
                                  price, orderDate);

            // Clear the text fields for quantity and order date.
            qtyDatePanel.clear();

            // Let the user know the order was placed.
            JOptionPane.showMessageDialog(null, "Order Placed.");
         }
```

```
123            catch (SQLException ex)
124            {
125               ex.printStackTrace();
126               System.exit(0);
127            }
128         }
129      }
130
131      /**
132         Private inner class that handles exit button events
133      */
134
135      private class ExitButtonListener implements ActionListener
136      {
137         public void actionPerformed(ActionEvent e)
138         {
139            System.exit(0);
140         }
141      }
142
143      /**
144         main method
145      */
146
147      public static void main(String[] args)
148      {
149         new PlaceOrder();
150      }
151   }
```

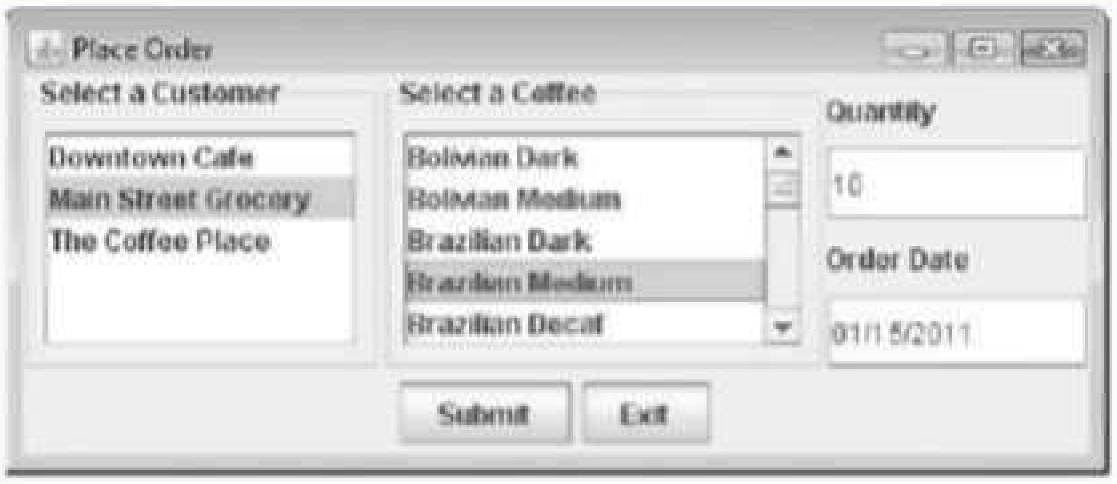
In the constructor, lines 24 through 27 set the `JFrame`'s title and specify an action for the close button. Lines 30, 33, and 36 create instances of the `CustomerPanel`, `CoffeePanel`, and `QtyDatePanel` classes. Line 39 calls the `buildButtonPanel` method. The `buildButtonPanel` method, which appears in lines 60 through 76, creates a panel with two `JButton` components: a Submit button and an Exit button. We will look at these buttons' event handlers in a moment. Back in the constructor, line 42 creates a `BorderLayout` manager. Lines 45 through 48 add the panels to appropriate regions of the content pane. Lines 51 and 52 pack and display the `JFrame`.

The `SubmitButtonListener` class, in lines 82 through 129, is the event handler for the Submit button. Line 89 retrieves the customer name from the `CustomerPanel` object. Line 92 retrieves the coffee description from the `CoffeePanel` object. Lines 95 and 98 retrieve the quantity and order date from the `QtyDatePanel` object. Line 101 creates an instance of the `CoffeeDBManager` class, which we will use to submit the order.

We have the name of the customer placing the order, and the name of the coffee being ordered, but to submit an order we need the customer number and the product number. Lines 104 and 105 call the `CoffeeDBManager` object's `getCustomerNum` method to retrieve the customer number. Line 108 calls the `CoffeeDBManager` object's `getProdNum` method to retrieve the product number. We also need the price of the coffee. Line 111 calls the `CoffeeDBManager` object's `getCoffeePrice` method to retrieve this information. Lines 114 and 115 call the `CoffeeDBManager` object's `submitOrder` method to submit the order. After the order is submitted, line 118 clears the text fields holding the quantity and order date, making it easier to enter the next order. Line 121 displays a dialog box indicating that the order was placed.

Figure 16-25 shows the order entry GUI with a customer selected, a coffee selected, a quantity entered, and an order date entered.

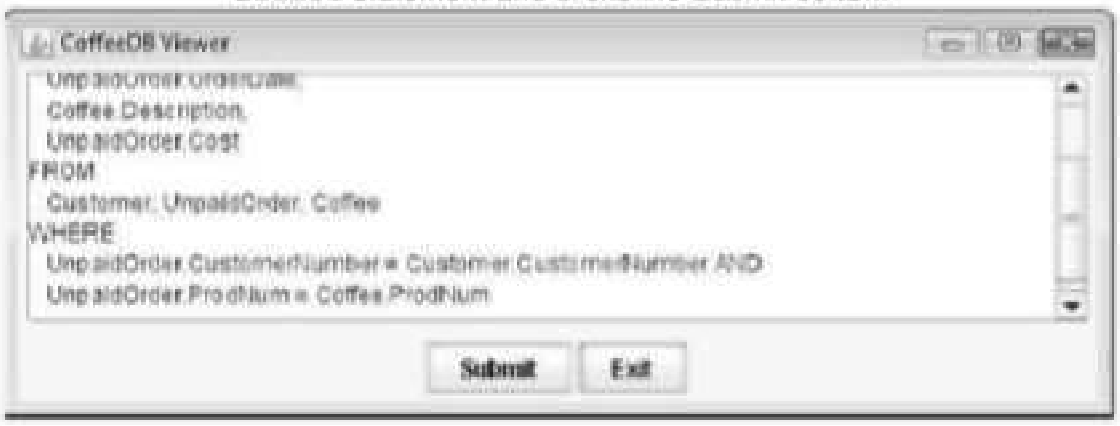
Figure 16-25 Order data entered



After we submit the order shown in Figure 16-25, we can run the `CoffeeDBViewer` application and enter the following `SELECT` statement to pull data from various tables relating to the order. Figure 16-26 shows the `CoffeeDBViewer` application's opening screen with the `SELECT` statement already filled in, and the results of the statement.

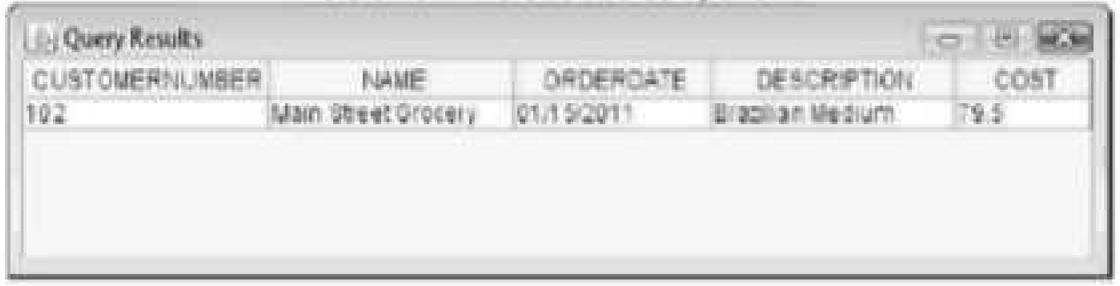
Figure 16-26 Order information viewed in `CoffeeDBViewer`

This window appears first in the `CoffeeDBViewer` application. The user enters a `SELECT` statement and clicks the Submit button.



```
  UnpaidOrder.OrderDate,
  Coffee.Description,
  UnpaidOrder.Cost
FROM
  Customer, UnpaidOrder, Coffee
WHERE
  UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
  UnpaidOrder.ProdNum = Coffee.ProdNum
```

This window appears next. It displays the results of the `SELECT` statement in a `JTable` component.



CUSTOMERNUMBER	NAME	ORDERDATE	DESCRIPTION	COST
102	Main Street Grocery	01/15/2011	Brazilian Medium	79.5

```
SELECT
   Customer.CustomerNumber,
   Customer.Name,
   UnpaidOrder.OrderDate,
   Coffee.Description,
   UnpaidOrder.Cost
FROM
   Customer, UnpaidOrder, Coffee
WHERE
   UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
   UnpaidOrder.ProdNum = Coffee.ProdNum
```

16.12 Advanced Topics

Transactions

Sometimes an application must perform several database updates to carry out a single task. For example, suppose you have a checking account and a car loan at your bank. Each month, your car payments are automatically taken from your checking account. For this operation to take place, the balance of your checking account must be reduced by the amount of the car payment, and the balance of the car loan must also be reduced.

An operation that requires multiple database updates is known as a *transaction*. In order for a transaction to be complete, all of the steps involved in the transaction must be performed. If any single step within a transaction fails, then none of the steps in the transaction should be performed. For example, imagine that the bank system has begun the process of making your car payment. The amount of the payment is subtracted from your checking account balance, but then some sort of system failure occurs before the balance of the car loan is reduced. You would be quite upset to learn that the amount for your car payment was withdrawn from your checking account, but never applied to your loan!

Most database systems provide a means for undoing the partially completed steps in a transaction when a failure occurs. When you write transaction-processing code, there are two concepts you must understand: commit and rollback. The term *commit* refers to making a permanent change to a database, and the term *rollback* refers to undoing changes to a database.

By default, the JDBC `Connection` class operates in auto commit mode. In *auto commit* mode, all updates that are made to the database are made permanent as soon as they are executed. When auto commit mode is turned off, however, changes do not become permanent until a commit command is executed. This makes it possible to use a rollback command to undo changes. A rollback command will undo all database changes since the last commit command.

In JDBC, you turn auto commit mode off with the `Connection` class's `setAutoCommit` method, passing the argument `false`. Here is an example:

```
conn.setAutoCommit(false);
```

You execute a commit command by calling the `Connection` class's `commit` method, as shown here:

```
conn.commit();
```

A rollback command can be executed with the `Connection` class's `rollback` method, as shown here:

```
conn.rollback();
```

Let's look at an example. Suppose we add a new table named `Inventory` to the `CoffeeDB` database, for the purpose of storing the quantity of each type of coffee in inventory. The table has two rows: `ProdNum`, which is a coffee product number, and `Qty`, which is the quantity of each type of coffee. When an order is placed, we want to update both the `Inventory` table and the `UnpaidOrder` table. In the `Inventory` table we want to subtract the quantity being ordered from the quantity in inventory. In the `UnpaidOrder` table we want to insert a new row representing the order. Here is some example code that might be used to process this as a transaction.

```
Connection conn = DriverManager.getConnection(DB_URL);
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
// Attempt the transaction.
try
{
   // Update the inventory records.
   stmt.executeUpdate("UPDATE Inventory SET Qty = Qty - " +
                 qtyOrdered + " WHERE ProdNum = " + prodNum);
   // Add the order to the UnpaidOrder table.
   stmt.executeUpdate("INSERT INTO UnpaidOrder VALUES('" +
                     custNum + "', '" +
                     prodNum + "', '" + orderDate + "', " +
                     qtyOrdered + ", " + cost + ")");
   // Commit all these updates.
   conn.commit();
}
catch (SQLException ex)
{
   // Roll back the changes.
   conn.rollback();
}
```

Notice that inside the try block, after the statements to update the database have been executed, the `Connection` class's `commit` method is executed. In the catch block, the `rollback` method is executed in the event of an error.

Stored Procedures

Many commercial database systems allow you to create SQL statements and store them in the DBMS itself. These SQL statements are called *stored procedures*, and they can be executed by other applications using the DBMS. If you have written an SQL statement that is