```
...
 // Create an object for collection of employees
 ArrayList<EmployeeDTO> employees = new ArrayList<EmployeeDTO>();

 // Process ResultSet and populate the collection

 while (rs.next()){
  EmployeeDTO currentEmp = new EmployeeDTO();
  currentEmp.setEmpNo(rs.getInt("EMPNO"));
  currentEmp.setEName(rs.getString("ENAME"));
  currentEmp.setJob(rs.getString("JOB_TITLE"));

  employees.add(currEmp);
 }
...
}
```

If this code is deployed on the server's JVM and you need to send the data to another computer that runs, say, a Swing client, you can consider applying Java serialization here for sending a collection of employees to the front. But make sure that the class `EmployeeDTO` implements the `Serializable` interface.

The PreparedStatement Class

Listing 22-1 uses the class `Statement` to create an object capable of executing SQL. But this is not the only way to supply SQL to the JDBC API. The class `PreparedStatement` is a subclass of `Statement`, but it pre-compiles the SQL statement before executing it.

With `PreparedStatement` you can create SQL with parameters that are dynamically passed by the program. Suppose you need to execute the query `"SELECT * from EMP WHERE empno=..."` multiple times, providing the `empno` values from the array `empNumbers[]`. If you use the class `Statement`, the variable `sqlQuery` will have to be pre-compiled on each iteration of the loop:

```
for (int i=0;i<empNumbers.length; i++){
```

```
  sqlQuery="SELECT * from Employee WHERE empno=" + employees[i];
  stmt.executeQuery(sqlQuery);
}
```

The class `PreparedStatement` offers a more efficient solution:

```
PreparedStatement stmt=conn.prepareStatement(
                       " SELECT * from Employee WHERE empno=?");

for (int i=0;i<empNumbers.length; i++){

  // pass the array's value that substitutes the question mark
  stmt.setInt(1,employees[i];)
  stmt.executeQuery(sqlQuery);
}
```

In this case the SQL statement is compiled only once and parameters are provided by the appropriate `setXXX()` method depending on the data type. The SQL statement may have several parameters (question marks), and the first argument of the setter enables you to specify each parameter's number. For example:

```
PreparedStatement stmt=conn.prepareStatement(
      "SELECT * from Employee WHERE empno=? and ename=?");

for (int i=0;i<empNumbers.length; i++){
  stmt.setInt(1,empNumbers[i];)
  stmt.setString(2,empNames[i];)
  stmt.executeQuery(sqlQuery);
}
```

If you need to pass a `NULL` value as a parameter, use the method `setNull()`.

The CallableStatement Class

This class extends `PreparedStatement` and is used for executing stored procedures from Java. Let's say there is a stored procedure entitled `changeEmpTitle` that takes two parameters: `empno` and `title`. Here's the code to execute this stored procedure:

```
CallableStatement stmt = conn.prepareCall("{call changeEmpTitle(?,?)}");

stmt.setInt(1,7566);
stmt.setString (2,"Salesman");
stmt.executeUpdate();
```

If a stored procedure returns some values using output parameters, each of the `OUT` data types has to be registered before the statement is executed. The next code snippet shows you an example of executing a stored procedure that has two parameters: The first is an input parameter, and the second is an output parameter by which the stored procedure can return the result of its execution to the Java program:

```
CallableStatement stmt = conn.prepareCall(
                 ("{call getEmpTitle(?,?) }");
stmt.setInt(1, 7566);
stmt.registerOutParameter(2,java.sql.Types.VARCHAR);
stmt.executeQuery();
String title=stmt.getString(2);
```

The ResultSetMetaData Class

JDBC enables you to process result sets when the number of returned values is unknown. Imagine that you need to write a program that can accept any SQL `Select` statement, execute it, and display the retrieved

data. With the class `ResultSetMetaData` you can dynamically find out how many columns there are in the result set, as well as their names and data types. The following code fragment gets the number of the database table columns in the result set and for each of them identifies and prints the column name and type:

```
String sqlQuery = "select * from Employee";
ResultSet rs = stmt.executeQuery(query);

ResultSetMetaData rsMeta = rs.getMetaData();
int colCount = rsMeta.getColumnCount();

for (int i = 1; i <= colCount; i++)  {
  System.out.println(
      " Column name: " + rsMeta.getColumnName(i) +
      " Column type: " + rsMeta.getColumnTypeName(i));
}
```

This simple but powerful technique is used internally by ORM frameworks (see Lesson 36) that can “magically” generate database models and automatically generate Java classes representing database entities.

Listing 22-3 shows a Java program called ShowAnyData that prints a result set based on any SQL `Select` statement passed from the command line. For example, it can be started as follows:

```
java ShowAnyData "Select * from Employee"
```



Listing 22-3: Using ResultSetMetaData

```
import java.sql.*;
class ShowAnyData {

 public static void main(String args[]) {
   Connection conn=null;
   Statement stmt=null;
   ResultSet rs=null;

 if (args.length==0){

    System.out.println("Usage: java ShowAnyData SQLSelectStatement");
    System.out.println("For example: java ShowAnyData \"Select * from Employee\"");
    System.exit(1);
  }

   try {
                  // Class.forName("org.apache.derby.jdbc.ClientDriver");
      conn = DriverManager.getConnection(
                           "jdbc:derby://localhost:1527/Lesson22");

      stmt = conn.createStatement();

    // Execute SQL query passed from command line
    rs = stmt.executeQuery(args[0]);

    // Find out the number of columns, their names and display the data
    ResultSetMetaData rsMeta = rs.getMetaData();
    int colCount = rsMeta.getColumnCount();

     for (int i = 1; i <= colCount; i++)  {
      System.out.print(rsMeta.getColumnName(i) + " ");
    }

    System.out.println();

    while (rs.next()){
       for (int i = 1; i <= colCount; i++)  {
         System.out.print(rs.getString(i) + " ");
       }
       System.out.print("\n");   // new line character
    }

   } catch( SQLException se ) {
      System.out.println ("SQLError: " + se.getMessage ()
            + " code: " + se.getErrorCode ());
```

```
    } catch( Exception e ) {
       System.out.println(e.getMessage());
       e.printStackTrace();
    } finally{

        // clean up the system resources
        try{
                     rs.close();
                     stmt.close();
                     conn.close();
        } catch(Exception e){
            e.printStackTrace();
        }
    }
  }
}
```

The output of the ShowAnyData program will be the same as that of EmployeeList shown in **Listing 22-1**. But this program can execute any SQL statement as long as you are specifying valid database objects. Note that the code in ShowAnyData first ensures that you have passed the command-line argument. If you run this program from a command line, don't forget to include the SQL statement in double quotes. In Eclipse you can specify a command-line argument by selecting the Run Configuration panel in the Arguments tab.

Scrollable Result Sets and RowSet

In all the preceding examples the code traversed the result set using the method `next()`, which moves only the cursor forward. Another option is to create a scrollable result set so the cursor can be moved back and forth if need be. There is a two-argument version of the method `createStatement()`. The first argument specifies the type of scrolling (`TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE`) and the second makes the result set updateable or read-only (`CONCUR_READ_ONLY` or `CONCUR_UPDATABLE`). For example,

```
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * from Employee");
```

The `TYPE_FORWARD_ONLY` parameter allows only forward movement of the cursor. The difference between `TYPE_SCROLL_INSENSITIVE` and `TYPE_SCROLL_SENSITIVE` is in whether scrolling reflects changes that have been made to the result set. The next example sets the cursor at the end of the result set and moves the cursor backward:

```
rs.afterLast();
while (rs.previous()){
 int empNo = rs.getInt("EMPNO");
 String eName = rs.getString("ENAME");
 String job = rs.getString("JOB");
 System.out.println(""+ empNo + ", " + eName + ", " + job_title);
}
```

You can also move the cursor to a specific row by using the following self-explanatory methods:

```
rs.absolute(25);  // moves the cursor to the 25th row
rs.relative(-4);  // moves the cursor to the 21st row
rs.first();
rs.last();
rs.beforeFirst();
```

If the result set is updatable (`CONCUR_UPDATABLE`) you can modify the underlying database table while scrolling. For example, the following state-

ments will update the job title of the employee based on the current cursor's position:

```
rs.updateString("JOB_TITLE","Manager");
rs.updateRow();
```

Scrollable result sets enable you to traverse the result set in both directions, but they have a drawback: They hold the database connection, which may be required by another thread or program. The package `javax.sql` includes the interface `RowSet`, which is a subclass of `ResultSet`. `RowSet` gets the data from the database, then disconnects, but still allows Java to work with the data. The package `javax.sql.rowset` has several concrete classes that implement `RowSet`, such as `CachedRowSet`, `FilteredRowSet`, and `WebRowSet`. The latter can turn `RowSet` into an XML stream to be sent to another tier in the distributed application.

Transactional Updates

Sometimes several database modifications have to be processed as one transaction, and if one of the updates fails, the whole transaction has to be **rolled back**. These database operations have to be explicitly **committed** (finalized) in case of success. If you set the auto-commit parameter on the database connection to `false`, the database transaction will not be committed until the code explicitly calls the method `commit()`, as in the following example:

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

  Statement stmt = con.createStatement();

  stmt.addBatch("insert into Orders " +
```

```
                "values(123, 'Buy','IBM',200)");
    stmt.addBatch("insert into OrderDetail " +
                "values('JSmith', 'Broker131', '05/20/02')");
    stmt.executeBatch();

    conn.commit();    // Transaction succeded

  }catch(SQLException e){
    conn.rollback();  // Transaction failed
    e.printStackTrace();
  }
```

In the preceding code snippet two `Insert` statements have to be executed as one transaction, and if any of them fails, an exception will be thrown and the method `rollback()` will undo all the changes, including those that succeeded.

Connection Pools and DataSources

Up until now you've been running all sample Java programs on your own computer. But imagine a distributed application in which multiple clients make requests to the same server, which has to process their SQL queries. Because obtaining a connection to the database is a slow process, it would be very inefficient to start every SQL request by obtaining a database connection and disconnecting after the request is complete. Such applications should reuse the same opened connection for multiple requests.

The package `javax.sql` includes the interface `DataSource`, which is an alternative to `DriverManager`. Vendors of JDBC drivers for servers implement this interface, and a `DataSource` is typically preconfigured for a certain number of connections (the **connection pool**). It is published in a directory using the JNDI interface. In such a setup, all clients' requests will get their database connections from this `DataSource` object, eliminating the need to open and close a new connection for each request. I'll provide an example of working with `DataSource` objects in Lesson 31 after explaining how JNDI works.

Try It

In this assignment you'll modify the class `Portfolio` from Lesson 22, which was just printing some hard-coded statements. Now you'll create and populate the database table Portfolio and then read and display the data from there.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at www.wrox.com. You can find them in the Lesson22 folder in the download.

Hint

Obtaining a database connection is a slow operation, and doing it from inside the method `run()` every time you start a new thread is not the best solution. Consider creating a database connection up front and passing it to the thread before starting it.

Step-by-Step

1. In the database for Lesson 22, create the table Portfolio using the following SQL statement:

```
create table Portfolio(
id NUMBER NOT NULL,
symbol VARCHAR2(10) NOT NULL,
quantity VARCHAR2(10) NOT NULL,
price NUMBER NOT NULL, PRIMARY KEY (id)
);
```

2. Populate the table Portfolio with three records, for stocks traded under the symbols IBM, AMZN, and AAPL respectively:

```
insert into Portfolio  values (1,'IBM',500,105.50),
 (2,'AMZN',1000,15.25),(3,'AAPL',2000,272.50);
```

3. Create a new Eclipse project called Lesson22.

4. Create a class called `Portfolio` that is similar to the one shown in Listing 20-5, but instead of just printing "You have 500 shares of IBM," have it connect to the database, select all the data from the table `Portfolio`, and print the symbol, quantity, and total value. (Calculate total value by multiplying price by quantity.)

5. Create a testing class called `ShowMyPortfolio` that instantiates and starts the thread `Portfolio`.

6. Test this program.



Please select Lesson 22 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 23

Swing with JTable

In this lesson you'll learn how to work with a Swing control that is advanced and popular for enterprise applications: `JTable`. This UI component enables you to present data in a grid with rows and columns. After learning the basics of working with `JTable`, in the Try It section you'll apply these new skills to display the portfolio data that, as of Lesson 22, is stored in the database.

In other words, you'll build a client-server application, where the Swing-based GUI is a client and the RDBMS is a server. Such architecture was pretty popular in the mid-nineties. Rich clients were developed in Visual Basic, PowerBuilder, Delphi, Java, or C++, and they connected directly to database servers such as Oracle, DB2, Sybase, Microsoft SQL Server, and Informix.

In the late nineties, thin clients (plain-looking HTML-based web pages with almost no code implementing business logic) became the trend. These days applications with rich UIs are coming back, but typically you'll be using an application server as a middleman between the client and the data storage. I'll describe such middlemen starting in Lesson 26, but your UI skills would be incomplete without your knowing how to program data grids.

JTable and the MVC Paradigm

The Swing class `JTable` is a powerful UI component created for displaying tabular data like a spreadsheet. The data is represented as rows and columns; that's why the `JTable` component is often used to display data from relational databases, which store data similarly. `JTable` was designed according to the **Model-View-Controller (MVC)** design pattern, according to which components responsible for presentation (or the view) are separated from components that store data (or the model) for that presentation. In general, design patterns suggest language-independent solutions for common programming tasks.

`JTable` is responsible for the visible portion of the grid (the V part of MVC), but the data has to be stored in a different Java class that implements the `TableModel` interface (the M part). Any other UI component can play the role of the controller (the C part) and initiate some actions that will move the data from model to view or vice versa. For example, a click on the `JButton` can initiate the population of the table model from the database and display the data in `JTable`.

The Model

Swing includes the classes `DefaultTableModel` and `AbstractTableModel`, which implement the `TableModel` interface and have methods to notify a `JTable` when the data is changing.

A programmer usually creates a model as a subclass of `AbstractTableModel`, and this class has to contain the data in some collection, for example `ArrayList`. When `JTable` needs to be populated, it requests the data from a class that implements `TableModel`, invoking such callback methods as `getColumnCount()` and `getValueAt()`. When a Swing program creates an instance of `JTable` it has to assign to it a corresponding table model class. **Listing 23-1** shows how the class `MyTableModel` (created by you) is given to the constructor of `JTable`.

Typically, the UI class that creates `JTable` defines one or more listeners that are notified of any changes in the table's data. The incomplete

class `MyFrame` in **Listing 23-1** implements the `TableModelListener` interface that defines just one method - `tableChanged()`. This method should contain the code performing data modifications, for example code to save the data in the database.



Listing 23-1: A window with JTable

```
package com.practicaljava.lesson23;

import javax.swing.*;
import javax.swing.table.AbstractTableModel;
import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;

public class MyFrame  extends JFrame implements TableModelListener{

  private MyTableModel myTableModel;
  private JTable myTable;

  MyFrame (String winTitle){
  super(winTitle);

  myTableModel = new MyTableModel();
  myTable = new JTable(myTableModel );

  //Add the JTable to frame and enable scrolling
  add(new JScrollPane( myTable));

  // Register an event listener
   myTableModel.addTableModelListener(this);
 }

 public void tableChanged(TableModelEvent e) {
  // Code to process data changes goes here
 }

 public static void main(String args[]){
  MyFrame myFrame = new MyFrame( "My Test Window" );

  myFrame.pack();
```

```
    myFrame.setVisible( true );

  }

    class MyTableModel extends AbstractTableModel {
      // The data for JTable should be here
    }

  }
```

In very simple cases you can create a `JTable` without declaring a table model class (`JTable` has a no-argument constructor), but Java will internally use its `DefaultTableModel` class anyway. My sample class `MyFrame`, though, will use the data model that's a subclass of the `AbstractTableModel`.

Note that the class `MyTableModel` is an inner class declared inside the class `MyFrame`. Having a model as an inner class is not a must, but if the data model will be used with only one specific `JTable`, its model might as well be declared inside the same class.

The code in **Listing 23-1** is not complete; it doesn't include any data yet, and the table model must include the mandatory callbacks described in the next section.

Mandatory Callbacks of Table Models

The class that implements the `TableModel` interface and feeds data to `JTable` must include at least three callback methods: `getColumnCount()`, `getRowCount()`, and `getValueAt()`.

The method `getColumnCount()` must return an integer value — the number of columns in this `JTable`. This method will be called once by a `JTable`. For example, if you are planning to display orders, each of which consists of four fields — order ID, stock symbol, quantity, and price — just put one line in this method:

```
        return 4;
```

The method `getRowCount()` must return an integer — it will also be called only once. The data has to be placed into an array or a data collection (such as `ArrayList MyData`) before it appears on the screen, and the code for this method could look like this:

```
       return myData.size(); //myData is an ArrayList in this sample
```

The method `getValueAt(int row, int col)` returns an `Object` and will be called once for each cell (each intersection of a row and a column). You have to write the code that will return the value for the requested row and column.

Let's say you have a class called `Order`, as shown in **Listing 23-2**, and you want to store instances of this class in `ArrayList myData`. This is going to be your data storage.



Listing 23-2: The Order class

```
public class Order {
 private int orderID;
 private String stockSymbol;
 private int quantity;
 private float price;

    public Order(int id, String stockSymbol, int quantity, float price){
```

```
            orderID=id;
            this.stockSymbol=stockSymbol;
            this.quantity=quantity;
            this.price=price;
        }
    }
```

Whenever the callback `getValueAt(int row, int col)` is called on the model, you have to return the cell value based on the passed row and column. The inner class `MyTableModel` from **Listing 23-3** includes the method `getValueAt()` working with `myData`, which is an `ArrayList` of `Order` objects.



Listing 23-3: The JFrame window with implemented table model

```
public class MyFrame  extends JFrame implements TableModelListener{

      MyTableModel myTableModel;
      JTable myTable;

 MyFrame (String winTitle){
  super(winTitle);

  myTableModel = new MyTableModel();
  myTable = new JTable(myTableModel );

  //Add the JTable to frame and enable scrolling
  add(new JScrollPane( myTable));

  // Register an event listener
   myTableModel.addTableModelListener(this);
 }

 public void tableChanged(TableModelEvent e) {
  // Code to process data changes goes here
 }

 public static void main(String args[]){
  MyFrame myFrame = new MyFrame( "My Test Window" );
```

```
  myFrame.pack();
  myFrame.setVisible( true );

}

// Inner class for data model
class MyTableModel extends AbstractTableModel {

 ArrayList<Order> myData = new ArrayList<Order>();

MyTableModel(){

      myData.add(new Order(1,"IBM", 100, 135.5f));
      myData.add(new Order(2,"AAPL", 300, 290.12f));
      myData.add(new Order(3,"MOT", 2000, 8.32f));
      myData.add(new Order(4,"ORCL", 500, 27.8f));
}

public int getColumnCount() {
 return 4;
}

public int getRowCount() {
 return myData.size();
}

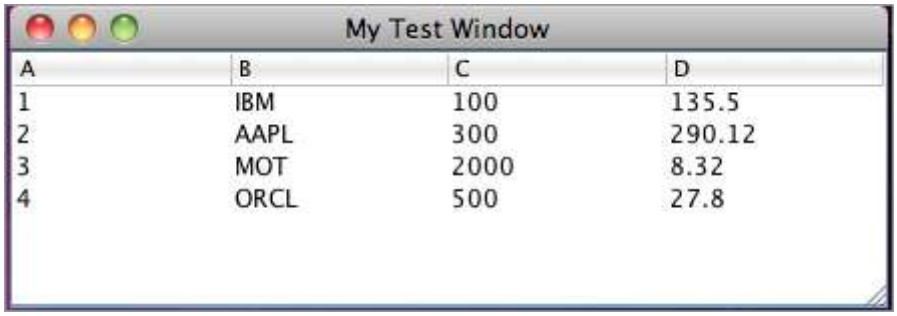
public Object getValueAt(int row, int col) {

      switch (col) {
        case 0:   // col 1
          return myData.get(row).orderID;
        case 1:   // col 2
              return myData.get(row).stockSymbol;
        case 2:   // col 3
              return myData.get(row).quantity;
        case 3:   // col 4
              return myData.get(row).price;
        default:
          return "";
      }
 }
}
}
```

Note the use of generics in the declaration of the `myData` collection. Another Java feature not to be missed here is **autoboxing** — the primitive `Order` fields `int` and `float` are automatically converted into the corresponding wrapper objects `Integer` and `Float`.

Running the program from **Listing 23-3** will display the window shown in **Figure 23-1**. Have you noticed that the window looks a bit unusual? The reason is that I decided to run this program on my MacBook laptop under the Mac OS. Java is a cross-platform language and this code will run without changes under Windows too — the look and feel will be different, though.

Figure 23-1



Optional Callbacks of Table Models

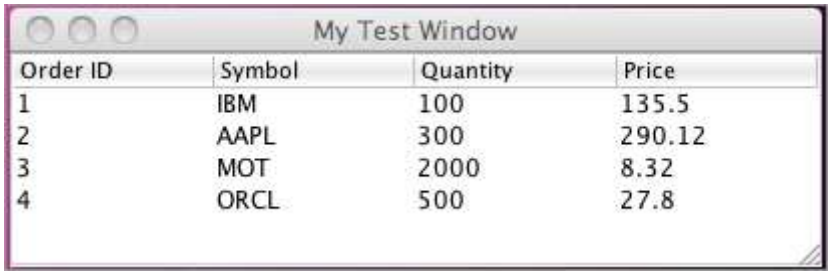
The `JTable` shown in **Figure 23-1** doesn't show the proper titles of the columns — the auto-generated A, B, C, and D don't count. You can fix this easily by overriding the `getColumnName()` method in the table model class. This callback, if present, will be called (once for each column) to render the column titles. Add the following code to the class `MyTableModel` and the window will look as it does in **Figure 23-2**:

```
 String[] orderColNames = { "Order ID",   "Symbol", "Quantity", "Price"};

public String getColumnName(int col) {
```

```
        return orderColNames[col];
    }
```

Figure 23-2



If you want to make some of the columns or cells editable, just override the `isCellEditable()` method and return `true` from this callback for the editable columns. Here's how to make the third column of our `JTable` editable:

```
public boolean isCellEditable(int row, int col) {

  if (col ==2){
      return true;
  } else {
    return false;
  }
}
```

If your table has editable columns you need to override the method `setValueAt(Object value, int row, int col)` and include the code that will copy the data from the UI component — `JTable` — to the appropriate field in its model objects. This method is called automatically when the user changes the value in a table cell and moves the cursor out of that cell by pressing the Enter or Tab key or by clicking a different cell.

The following method, `setValueAt()`, will take the modified order quantity and set the new value for the `quantity` field in the appropriate `Order` in the model. By default, all the data shown in `JTable`'s cell has the `String` data type, and it's your responsibility to do proper casting.

```
public void setValueAt(Object value, int row, int col){

  if (col== 2){
   myData.get(row).quantity=(Integer.valueOf(value.toString()));
  }

  //Notify listeners about the data change
   TableModelEvent event = new TableModelEvent(this, row, row, col);
   fireTableChanged(event);
}
```

The `fireTableChanged()` method has been placed in the `setValueAt()` method to notify any listener(s) who want to know about the data changes. For example, if the quantity on any order has been changed and has gone over a certain threshold, the application may need to immediately perform some actions to report this to some authority.

Review the code in **Listing 23-3**. The class `MyFrame` implements `TableModelListener`, so the method `tableChanged()` will be invoked as a result of the `fireTableChanged()` method call. Add the following line to the `tableChanged()` method:

```
System.out.println("Someone modified the data in JTable!");
```

Now run the program and modify the quantity in any row of `JTable`. The message will be printed on the system console. But JVM fires a table

changed event with a payload — `TableModelEvent` — that carries useful information about what exactly has changed in the table model.

`TableModelEvent` has several constructors; I've chosen the one that takes modified rows and columns as arguments. For example, if you change the quantity in the last row, as shown in **Figure 23-2**, the method `tableChanged()` will receive an instance of `TableModelEvent` that will encapsulate the reference to the entire model encapsulating the following values describing the change:

```
column=2    //  the third column
firstRow=3  //  starting from the row #4
lastRow=3   //  ending with the row #4
```

Based on this information you can implement any further processing required by the functional specification of your application. If you need to apply the UI changes to the database, the method `tableChanged()` can be the right place to use the JDBC API or other communication with the server-side code to persist the changes.

There are several functions with names that start with the word `fire`. For example, to apply each cell's change to the database, call the method `fireTableCellUpdated()`. To apply all changes at once, call the method `fireTableDataChanged()`. Refer to the documentation of the class `AbstractTableModel` to decide which method fits your needs.

Introduction to Renderers

The process of transferring data from the table model to `JTable` is performed by **cell renderers**. When the user is modifying the content of the cell, the **cell editor** is engaged. By default, the content of each cell in a `JTable` is rendered using one of three default renderers, based on the type of data in the column. Boolean values are rendered as check-

boxes, `javax.swing.Icon` is rendered as an image, and any other object is rendered as a string of characters.

To change the default rendering (for example, if you don't want to see checkboxes for Boolean values) you can either override the callback `getColumnClass()` or define a **custom cell renderer**. The latter option gives you a lot more flexibility. For example, you may need to display a photo of a person and his or her name in each cell. Or you may need to show cell values that meet certain criteria in a different color. To do something like one of these, you need to create a custom renderer.

The UI portion of each column is represented by the class `TableColumn`, which has a property, `cellRenderer`, of type `TableCellRenderer`, which defines the only method: `getTableCellRendererComponent()`. This method prepares the content for each column's cells of `JTable` and returns an instance of the `Component` class to be used for the cell rendering. This process will use `DefaultTableCellRenderer` unless you create a custom renderer. Custom renderers give you full control over how the cell is displayed.

The class `DefaultTableCellRenderer` extends `JLabel` and is Swing's implementation of the `TableCellRenderer` interface. Let's look at an example that formats the text in the Price column shown in **Figure 23-2** to be right-justified and to display in red all prices greater than $100.

First the code fragment from **Listing 23-4** gets a reference to the fourth column of `JTable` (remember, column numbering is zero-based). Then it needs to call the method `setCellRenderer()` on this column, provided that the custom renderer class was defined and instantiated. But you can define, instantiate, and set the custom renderer in one shot by using the mechanism of **anonymous inner classes**.

The anonymous inner class in **Listing 23-4** extends the class `DefaultTableCellRenderer` and overrides the callback method `getTableCellRendererComponent()`. The latter sets the cell value to be right-justified and to be

red if it is greater than 100. At the end, the method `getTableCellRendererComponent()` returns a `JLabel` object to be rendered in the current cell of `JTable`.



Listing 23-4: Custom rendering of the Price value

```
//Assign custom cell renderer to the Price column
// Get the reference to the fourth column - Price

TableColumn column = myTable.getColumnModel().getColumn(3);

// Create a new cell renderer as an anonymous inner
// class and assign it to the column price

column.setCellRenderer(
      new DefaultTableCellRenderer(){
  public Component  getTableCellRendererComponent(
          JTable table, Object value, boolean isSelected,
                    boolean hasFocus, int row, int col) {

       JLabel label = (JLabel) super.getTableCellRendererComponent(
             table, value, isSelected, hasFocus, row, col);

     // right-align the price value
     label.setHorizontalAlignment(JLabel.RIGHT);

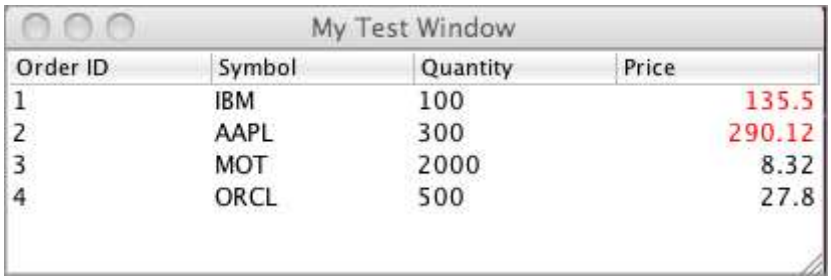
     // display stocks that cost more than $100 in red
     if (((Float) value)>100){
        label.setForeground(Color.RED);
     } else{
           label.setForeground(Color.BLACK);
     }

     return label;

    } // end of getTableCellRendererComponent
   }  // end of new DefaultTableCellRenderer
);    // end of setCellRenderer(...)
```

Add this code fragment at the end of the constructor in the class `MyFrame` from **Listing 23-3** and run the application. The screen will show the text in the Price column right-justified and the first two prices printed in red (see **Figure 23-3**).

Figure 23-3



Summary

This lesson was a high-level overview of the `JTable` component, which is probably the most advanced UI component that deserves serious study if you are planning to develop Java Swing applications. You can continue studying all the features of `JTable` by following the official Oracle tutorial at

`http://download.oracle.com/javase/tutorial/uiswing/components/table.html`.

Try It

Create an application that will display your portfolio, stored in the database table. You'll need to use the database and the Portfolio table you created in the Try It section of Lesson 22.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find

them in the Lesson23 folder in the download.

Hint

In Windows, the Java installation directory includes several examples of `JTable`, which can serve as good complementary material to this lesson. Just go to the folder `demo\jfc\TableExample` and follow the instructions in the `Readme` document.

Step-by-Step

1. Create the Eclipse project Lesson23. Copy there the classes `MyFrame` and `Order` from the accompanying DVD. Compile and run the program to ensure that it displays hard-coded portfolio data, as shown in **Figure 23-2**.

2. Replace the hard-coded table model `ArrayCollection myData` with the JDBC code to connect to the database Lesson22, run the appropriate SQL `Select` statement, and populate `myData` with data from the received result set.

3. Test this program.



Please select Lesson 23 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 24

Annotations and Reflection

In general, **metadata** is data about your data. In the context of DBMSes, metadata can be information describing the way you store data, such as the table and field names or primary keys. Program code metadata is data about your code. Any Java class has its metadata embedded, and you can write a program that "asks" another class, "What methods do you have?" or similar questions about class fields, constructors and ancestors.

If you've ever looked at the source code of any Java class, you can easily identify Javadoc-style comments for classes, interfaces, variables, and methods. These comments may include specially formatted words.

In Eclipse you can select any Java class and press F3 to open the source code of this class. Because the previous lesson was about working with `JTable`, let's open the source code of this class. In the top part of the code you'll find a description similar to the one that follows (I removed a large portion of the text for brevity):

```
/**
 * The <code>JTable</code> is used to display and edit regular two-dimensional
 * tables of cells.
 * To enable sorting and filtering of rows, use a
 * {@code RowSorter}.
 *  * As for all <code>JComponent</code> classes, you can use
 * {@link InputMap} and {@link ActionMap} to associate an
 * {@link Action} object with a {@link KeyStroke} and execute the
 * action under specified conditions.
 * <p>
 * <strong>Warning:</strong> Swing is not thread safe. For more
 * information see <a
 * href="package-summary.html#threading">Swing's Threading
 * Policy</a>.
 * <p>
 *
 * @version 1.292 05/30/08
 * @author Philip Milne
 * @author Shannon Hickey (printing support)
 * @see javax.swing.table.DefaultTableModel
 * @see javax.swing.table.TableRowSorter
 */
```

The special words marked with an `@` sign are Javadoc metadata describing links, version, author, and related classes to see in Java documentation. If you run the source code of this class through the Javadoc utility the utility will generate HTML output that can be opened by any web browser. It's a good practice to include Javadoc comments in your classes.

Javadoc acts as a processing tool that extracts from the source code all comment blocks that start with `/**` and end with `*/`. It then formats this text using HTML tags and embedded annotations to generate program documentation. The preceding text is an example of the use of specific tags that are predefined, and understood by Javadoc. This was an example of metadata that are understood by just one utility — Javadoc.

But Java allows you to declare your own custom annotations and define your own processing rules that will route the execution of your program and produce configuration files, additional code, deployment descriptors, and more. No matter what your goals, when you create annotations you also need to create or use an **annotation processor** to get the expected output.

Starting in Lesson 27 you'll learn about annotations defined by the creators of Java EE technologies. But in this lesson you'll get familiar with Java SE annotations, which you will eventually use in your projects.

Java Annotations Basics

The Java annotation model enables you to add custom annotations anywhere in your code, not just in the comments section. You can apply custom annotations to a class, a method, or a variable — just specify allowed **targets** when the annotation is being defined. Java annotations start with the `@` sign and may optionally have one or more parameters. Some of the annotations are built into Java SE and are used by the javac compiler, but most of them are consumed by some kind of processing program or tool.

There are about a dozen of predefined annotations already included with Java SE. You can find them, along with their supporting classes, in the packages `java.lang`, `java.lang.annotation`, and `javax.annotation`. You can check the Annotations section of the latest online documentation on the Java SE API, which at the time of this writing is located at http://download.oracle.com/javase/7/docs/api/overview-summary.html.

Some of these annotations are used by the compiler (`@Override`, `@SuppressWarning`, `@Deprecated`, `@Target`, `@Retention`, `@Documented`, and `@Inherited`); some are used by the Java SE run time or third-party run times and indicate methods that have to be invoked in a certain order (`@PostConstruct`, `@PreDestroy`), or mark code that was generated by third-party tools (`@Generated`). I'm not going to give you a detailed description of how to use each of these annotations, but I will give you selected examples that will help you get started with annotations.

@Override

In the Try It section of Lesson 3 you overrode the method `public double calcTax()` in the class `NJTax`. The method signature of `calcTax()` was the same in both `NJTax` and its superclass `Tax`. Now deliberately add an argument to `calcTax()` in NJTax, as if you had done so by accident. The code will compile with no errors. But you could have done this by mistake, and instead of overriding the method as planned, you've overloaded it. This won't hap-

pen if you use the annotation `@Override` whenever you are planning to override a method:

```
@Override public double calcTax(String something)
```

Now the compiler will complain with the following error:

```
The method calcTax(String) of type NJTax must override or implement a supertype method
```

The annotation `@Override` signals the compiler that overriding is expected, and that it has to fail if an override does not occur.

@SuppressWarning

The compiler can generate warning messages that won't stop your program from running, but that do indicate potential problems. In some cases, though, you want to suppress some or all warnings so the output of the project build looks clean. For example,-`Xlint:none` disables all warnings, while -`Xlint:fallthrough` instructs the compiler to warn you about if you forget to add the `break` statement to the `switch` statement (see Lesson 5). In Eclipse IDE, to set the compiler's options you right-click the project and select Properties ⇒ Java Compiler ⇒ Errors/Warnings ⇒ Potential Programming Problems.

But what if you want to omit the `break` keyword in the `switch` statement on purpose? You still want to get warned in all other cases about missing `break`, but not in this particular method. This is where the `@SupressWarnings` annotation becomes quite handy and **Listing 24-1** illustrates it. To see this example at work, turn on the compiler's option that warns you about the `switch case` fall-throughs.



Listing 24-1: Custom rendering of the Price value

```
package com.practicaljava.lesson24;

public class SuppressWarningDemo {

      @SuppressWarnings("fallthrough")
      public static void main(String[] args) {
            int salaryBand=3;
        int bonus;
       // Retrieve the salary band of the person from some data source here
            switch(salaryBand){
             case 1:
                  bonus=1000;
                  System.out.println("Giving bonus " + bonus);
                  break;
             case 2:
                  bonus=2000;
```

```
                System.out.println("Giving bonus " + bonus);
                break;
            case 3:
                bonus=6000;
                System.out.println("Giving bonus " + bonus);
            case 4:
                bonus=10000;
                System.out.println("Giving bonus " + bonus);
                break;
            default:
                // wrong salary band
                 System.out.println("Invalid salary band");
            }
        }

}
```

Note that the `break` keyword is missing in the `case 3` section. In this code it's done on purpose: All employees in `salaryBand` 3 are entitled to two bonuses — six thousand dollars in addition to ten thousand dollars. The compiler's annotation `@SuppressWarnings("fallthrough")` suppresses compiler warnings only for this method. In all other classes or methods that may have `switch` statements, the warning will be generated.

@Deprecated

If you are developing classes that are going to be used by someone else, mark as `@Deprecated` any classes or methods that may be removed in future versions of your code. Other developers will still be able to use this code, but they'll be warned that switching to newer versions is highly recommended.

@Inherited

This annotation simply means that the annotation has to be inherited by descendants of the class in which it is used. You'll see an example of its use in the next section.

@Documented

If you want an annotation to be included in the Javadoc utility, mark it as `@Documented`. You'll see an example of this annotation in the next section.

Custom Annotations

Creating your own annotations is more interesting than using core Java or third-party annotations. First of all, you have to decide what you need an annotation for and what properties it should have. Then you need to specify the allowed **targets** for this annotation (for example, class, method, or variable). Finally, you have to define your **retention policy** — how long and where this annotation will live. Let's go through an example to illustrate all these steps.

Suppose you need to create an annotation that will allow the user to mark class methods with a SQL statement to be executed during the run time. These classes will be loaded dynamically. The goal is to declare your own annotation to be used by other Java classes, and to write the annotation

processor that will read these Java classes, identify and parse annotations and the values of their parameters, if any, and do whatever is required accordingly.

Declaring annotations is very similar to declaring interfaces, but don't forget to add the `@` sign at the beginning. I'll name my annotation `MyJDBCExecutor`:

```
public @interface MyJDBCExecutor{

}
```

If metadata is data about data, then **meta-annotations** are annotations about annotations. This is not as confusing as it sounds. To specify where you can use this newborn annotation, define the meta-annotation `@Target`. The enumeration `ElementType` defines possible target values: `METHOD`, `TYPE`, `CONSTRUCTOR`, `FIELD`, `PARAMETER`, `PACKAGE`, `LOCAL_VARIABLE`, and `ANNOTATION_TYPE`. If you don't use `@Target`, the annotation can be used anywhere. For example, this is how you can allow use of the annotation only with methods and constructors:

```
import java.lang.annotation.*;

@Inherited
@Documented
@Target({ ElementType.METHOD, ElementType.CONSTRUCTOR })
@Retention(RetentionPolicy.SOURCE)
public @interface MyJDBCExecutor {

}
```

The retention policy in the preceding code snippet is set to `SOURCE`, which means that this annotation will be used for processing only during the compilation of your program. The other two allowed values for retention policy are `RUNTIME` and `CLASS`.

Annotations with the `CLASS` retention policy stay in the compiled class, but are not loaded during run time. The `CLASS` retention policy is used by default if a retention policy is not explicitly specified.

Annotations with the `RUNTIME` retention policy have to be processed by a custom processing tool when the compiled code is running.

Annotations may have parameters. Say we want to add a single parameter that will allow us to specify an SQL statement to be processed. Our `MyJDBCExecutor` has to be declared as follows (omitting the `import` statement):

```
@Target({ ElementType.METHOD, ElementType.CONSTRUCTOR })
@Retention(RetentionPolicy.SOURCE)
public @interface MyJDBCExecutor {
```

```
   String value();
}
```

A sample Java class, `HRBrowser`, may use this annotation like this:

```
class HRBrowser{

  @MyJDBCExecutor (value="Select * from Employee")
  public List getEmployees(){
     // add calls to some JDBC executing engine here
  }

}
```

If the annotation has only one parameter named `value`, the "`value="` part in the preceding code snippet is not required. But I'd like this annotation to have three parameters: SQL to execute, transactional support, and a notification flag to inform other users of the application about any database modifications. Let's add more parameters to our annotation:

```
package com.practicaljava.lesson24;

import java.lang.annotation.*;

@Target({ ElementType.METHOD})
@Retention(RetentionPolicy.SOURCE)
public @interface MyJDBCExecutor {
         String sqlStatement();
         boolean transactionRequired() default false;
         boolean notifyOnUpdates() default false;

}
```

I've replaced the parameter `value` with a more meaningful `sqlStatement`, and added two more: `transactionRequired` and `notifyOnUpdates`. I gave the latter two default values. If a Java class won't need to support transactions and notify other applications about updates, why force software developers to provide values for these parameters?

If we don't specify default values the Java compiler will generate compilation errors if the values for `transactionRequired` and `notifyOnUpdates` are missing in classes that use `@MyJDBCExecutor`. Here's an example of a class, `HRBrowser`, that's annotated only with a `select` statement — no other actions are needed here:

```
class HRBrowser{

  @MyJDBCExecutor (sqlStatement="Select * from Employee")
  public List<Employee> getEmployees(){
      // The code to get the the data from DBMS goes here,
      // result set goes in ArrayList, which is returned to the
```

```
        // caller of the method getEmployees()
        ...
        return myEmployeeList;
    }
}
```

The code sample in **Listing 24-2** adds the method `updateData()` and uses all three annotation parameters.



Listing 24-2: Using the annotation MyJDBCExecutor

```
class HRBrowser{

 @MyJDBCExecutor (sqlStatement="Select * from Employee")
 public List<Employee> getEmployees(){
        // Generate the code to get the the data from DBMS,
        // place them in ArrayList and return them to the
        // caller of my getEmployees
           ...
               return myEmployeeList;
 }

 @MyJDBCExecutor (
            sqlStatement="Update Employee set bonus=1000",
            transactionRequired=true,
            notifyOnUpdates=true)
 public void updateData(){
   // JDBC code to perform transactional updates  and
   // notifications goes here
 }
}
```

The annotations with the SOURCE retention policy can be processed by the Annotation Processing Tool (APT) located in the `bin` directory of your Java installation. As a result of such processing, the new source code may be generated for further compilation. (The use of APT is an advanced subject and is out of the scope of this tutorial. Refer to the following web page for more information: **http://download.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html**.)

I was involved in the development of an open-source code generator called Clear Data Builder (CDB) (available at **http://sourceforge.net/projects/cleartoolkit/**). The CDB allows the user to write a simple Java class that has an abstract method annotated with an SQL statement and several other parameters, and within seconds to generate complete code for the functional application that has Adobe Flex code on the client side talking to Java at the server, which is accessing data stored in any relational DBMS via JDBC. In this project we used only the annotations with the SOURCE retention policy, and before compiling classes would generate additional code according to specified annotations.

If CDB would be processing the annotation `@MyJDBCExecutor`, it would engage additional tools and generate and compile all JDBC code for the methods

`getEmployees()` and `updateData()` automatically.

For the annotations with the RUNTIME retention policy you should know how to write an annotation processor, however, as it has to "extract" the values from the annotations during run time, and, based on those values, engage the appropriate code. But there is one Java feature, **reflection**, that you must understand before you can write your own annotation-processing class.

Reflection

Strictly speaking the subject of Java reflection doesn't belong to the lesson on annotations — reflection is used widely in various areas of Java development. But because this subject has to be covered before you can proceed with annotation processing, I decided to include it here.

Reflection enables you to find out about the internals of a Java class (its methods, constructors, and fields) during the run time, and to invoke the discovered methods or access public member variables. A special class called `Class` can load the class in memory, and then you can explore the content of the class by using classes from the package `java.lang.reflect`. Consider the class `Employee`, shown in **Listing 24-3**.



Listing 24-3: Class Employee extends Person

```
abstract public class Person {
  abstract public void raiseSalary();
}

public class Employee extends Person{
 public void raiseSalary() {
   System.out.println("Raising salary for Employee...");
  }
}
```

The `ReflectionSample` class in **Listing 24-4** loads the class `Employee`, prints its method signatures, and finds its superclass and methods. The process of querying an object about its content during run time is called **introspection**.



Listing 24-4: Introspecting Employee

```
import java.lang.reflect.*;
public class ReflectionSample {
  public static void main(String args[]) {
     try {
       Class c = Class.forName("Employee");
       Method methods[] = c.getDeclaredMethods();
```

```
        System.out.println("The  Employee methods:");

        for (int i = 0; i < methods.length; i++){
             System.out.println("*** Method Signature:" +
                                methods[i].toString());
        }

        Class superClass = c.getSuperclass();
        System.out.println("The name of the superclass is "
                                + superClass.getName());

        Method superMethods[] = superClass.getDeclaredMethods();
        System.out.println("The superclass has:");

        for (int i = 0; i < superMethods.length; i++){
             System.out.println("*** Method Signature:" +
                               superMethods[i].toString());
             System.out.println("      Return type: " +
                 superMethods[i].getReturnType().getName());
        }

      } catch (Exception e) {
           e.printStackTrace();
      }
   }
}
```

Here's the output of the program ReflectionSample:

```
The  Employee methods:
*** Method Signature:public void Employee.raiseSalary()
The name of the superclass is Person
The superclass has:
*** Method Signature:public abstract void Person.raiseSalary()
      Return type: void
```

Some other useful methods of the class `Class` are `getInterfaces()`, `getConstructors()`, `getFields()`, and `isAnnotationPresent()`. The following code snippet shows how to get the names, types, and values of the public member variables of the loaded class:

```
Class c = Class.forName("Employee");

Field[] fields = c.getFields();
for (int i = 0; i < fields.length; i++)  {
   String name = fields[i].getName();
   String type = fields[i].getType().getName();

   System.out.println("Creating an instance of Employee");
   Object obj = c.newInstance();
   Object value = fields[i].get(obj);
   System.out.println("Field Name: " + name + ", Type: "
                + type + " Value: " + value.toString());
}
```

The process of reflection uses introspection to find out during run time what the methods (or properties) are, but it also can call these methods

(or modify these properties). The method `invoke()` will let you call methods that were discovered during run time:

```
Class c= Class.forName("Employee");
Method raiseSalary = c.getMethod( "raiseSalary", null);
raiseSalary.invoke(c.newInstance(),null);
```

The first argument of the method `invoke()` represents an instance of the object `Employee`, and `null` means that this method doesn't have arguments. With reflection, the arguments are supplied as an array of objects. You can find out what the method arguments are by calling the method `Method.getParameterTypes()`, or create and populate them on your own, as in the following example. Add the following method to the class `Employee`:

```
public void  changeAddress(String newAddress) {
    System.out.println("The new address is "+ newAddress);
}
```

Note the `public` qualifier: it's needed for proper introspection, otherwise the `NoSuchMethodException` will be thrown by the following code snippet. The `ReflectionSample` class can invoke `changeAddress()` as follows:

```
Class c= Class.forName("Employee");
Class parameterTypes[]= new Class[] {String.class};
Method myMethod = c.getMethod( "changeAddress", parameterTypes);

Object arguments[] = new Object[1];
arguments[0] = "250 Broadway";
myMethod.invoke(c.newInstance(),arguments);
```

Reflection helps in building dynamic component-based applications that can load different classes based on certain business logic and invoke this logic during run time. Many third-party Java frameworks read configuration files and then instantiate and use required objects.

Run-Time Annotation Processing

The author of a custom run-time annotation usually gives it to other developers along with the processing tool. They will add the annotation to their classes and compile them, and the processing tool will consume these classes during run time. Reuse the code example from **Listing 24-2**, but this time imagine that `@MyJDBCExecutor` becomes the annotation with `RUNTIME` retention policy and that there is no need to generate additional source code for the compilation time. Suppose this annotation will be used in `HRBrowser`, and another class will have to analyze the annotation parameters and route the execution accordingly.

Now I'll write the annotation processor class called `MyJDBCAnnotationProcessor`, and the class `HRBrowser` in **Listing 24-2** can serve as a command-line argument to that processor:

```
c:/>java MyJDBCAnnotationProcessor HRBrowser
```

The class `MyJDBCAnnotationProcessor` has to load the class `HRBrowser`, introspect its content, find the annotations and their values, and process them accordingly. I'll show you how to write such a processor, or rather its annotation-discovery part.

Listing 24-5 shows `MyJDBCAnnotationProcessor`, which starts by loading another class, whose name was supplied in the command line. After that it introspects the loaded class and places all references to its method definitions into an array called `methods`. Finally, it loops through this array, and for each method that has annotations it finds and prints the values of the parameters `sqlStatement`, `notifyOnUpdates`, and `transactionRequired`.



Listing 24-5: MyJDBCAnnotationProcessor

```
import java.lang.reflect.*;
import com.practicaljava.lesson24.MyJDBCExecutor;

public class MyJDBCAnnotationProcessor {

 public static void main(String[] args) {
  // TODO add a check for the number of command line arguments
  // has to be the name of the class to load.

  String classWithAnnotation = args[0];

    try {
       //Load provided on the command line class
      Class loadedClass = Class.forName(classWithAnnotation);

      // Get references to class methods
      Method[] methods = loadedClass.getMethods();

      // Check every method of the class.If the annotation is present,
      // print the values of its parameters

       for (Method m: methods){
        if (m.isAnnotationPresent(MyJDBCExecutor.class)){
                  MyJDBCExecutor jdbcAnnotation =
                            m.getAnnotation(MyJDBCExecutor.class);

          System.out.println("Method: " + m.getName() +
            ". Parameters of MyJDBCExecutor are: " +
            "sqlStatement="+ jdbcAnnotation.sqlStatement() +
            ", notifyOnUpdates="+ jdbcAnnotation.notifyOnUpdates() +
            ", transactionRequired="+ jdbcAnnotation.transactionRequired());
        }
      }

  } catch (ClassNotFoundException e) {
```

```
                    e.printStackTrace();
   }

  }
}
```

After running this processor with the class `HRBrowser`, the former correctly identifies the annotated methods and prints the values of their parameters:

```
Method: getEmployees. Parameters of MyJDBCExecutor are: sqlStatement=Select * from Employee, notifyOnUpdates=false, t

Method: updateData. Parameters of MyJDBCExecutor are: sqlStatement=Update Employee set bonus=1000, notifyOnUpdates=tr
```

Any class can use more than one annotation. In this case the annotation processor would need to start by getting all annotations of the loaded class using `loadedClass.getAnnotations()`. It would then process these annotations in a loop.

Summary

In real-world applications you wouldn't simply be printing the values of the annotation parameters, but rather would be executing different branches of your code based on these values. This is the point of run-time annotation processing. You may ask, "OK, now I know the annotations and their values, so what do I do with them?" The big idea is that you've written a generic processor that can work with any classes that include your annotations. It's a pretty powerful mechanism for all software developers who are creating tools for other people to use.

You'll probably be using annotations and run-time processors written by other people rather than ones you write yourself. You'll see lots of examples of using annotations starting from Lesson 27, while you're learning about Java EE development. But now that you know what's going on under the hood in annotation processors, learning about Java EE annotation will be a lot easier.

Try It

Create a class-level run-time annotation called `@DBParams` that will enable you to specify the name of the database, the user ID, and the password. Write a processor for this annotation.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson24 folder in the download.

Step-by-Step

1. Create the Eclipse project Lesson24.

2. Declare there the annotation `DBParams` with the retention policy `RUNTIME` targeted to `TYPE`.

3. Define three parameters in this annotation: `dbName`, `uid`, and `password`.

4. Create the class `MyDBWorker` and annotate it with `@DBParms` populated with some initial values.

5. Write an annotation processor class called `DBParamProcessor` to find and print the annotation values in the class `MyDBWorker`.

6. Run and test `DBParamProcessor`.



Please select Lesson 24 on the DVD with the print book, or watch online at **www.wrox.com/go/fainjava** to view the video that accompanies this lesson.

Chapter 25

Remote Method Invocation

So far most of the Java programs in this tutorial have been running in one JVM. There were two exceptions: In Lesson 18 you used two JVMs while learning about socket programming, and your JDBC programs from Lesson 22 communicated with another JVM running database server. The application running on the user's computer isn't always allowed to access remote data directly — that's one of the reasons **distributed Java applications** came into the picture. (The word distributed means having parts of the applications running on several computers.) The other reason was to provide a centralized server catering to multiple lightweight clients.

There are lots of ways to create Java distributed applications that run on more than one JVM, and Remote Method Invocation (RMI) is one of them even though it's being used seldom these days. For example, a client Java application (JVM1) connects to a server Java application (JVM2), which connects to the DBMS that runs on a third computer. The client application knows nothing about the DBMS; it gets the data, an `ArrayList` of `Employee` objects, from the server's application that runs in JVM2. RMI uses object serialization for the data exchange between JVM1 and JVM2.

But unlike with socket programming, where the client explicitly connects to the server, with RMI one Java class can invoke methods on Java objects that live in another (remote) JVM. Although from a syntax perspective it looks as if the caller and the server's class are located in the same JVM, they may be thousands of miles away. The RMI client won't have a copy of the server-side method; it'll just have the method's local representative — a **proxy**, or, using the RMI terminology, a **stub**.

Any RMI application consists of an RMI server, a client, and the **registry** (a naming service). These three components could run on three different JVMs running on different networked computers. The RMI server creates Java objects that implement business logic, registers them with the naming service, and waits for remote clients to invoke methods on them.

A client application gets a reference to a remote server object or objects from the registry and then invokes methods on this remote object. The main concept of RMI is that even though the methods are called in the client's JVM, they are executed on the server's! RMI supporting classes and the registry tool are included with the Java SE.

Developing Applications with RMI

This lesson is written as an illustration of a sample RMI application with a minimum theory. For a more detailed description of RMI please refer to the following website: **http://download.oracle.com/javase/6/docs/technotes/guides/rmi/index.html**.

Writing distributed RMI applications involves the following steps:

1. Declaring the remote interface.

2. Implementing the remote interface.

3. Writing a Java client that connects to the remote server and calls remote methods.

4. Starting the registry and registering the RMI server with it.

5. Starting the server and the client applications.

Let's perform each of these steps by developing the RMI version of the Stock Quotes Server (see its version with sockets in Lesson 18), which will provide a client with price quotes for a specified stock. Some of the following steps could be combined, but for simplicity's sake I'll write a separate class for each required action.

Defining Remote Interfaces

The Java classes that you are planning to deploy on the server side have to implement **remote interfaces**, which declare business method(s) to be invoked remotely by RMI clients. The client's code will look as if it's calling local methods, but these calls will be redirected to a remote server via the RMI protocol. Following are the rules for creating remote interfaces:

- The remote interface must declare public methods to allow clients to perform remote method invocation.
- The remote interface must extend `java.rmi.Remote`.
- Each method must declare `java.rmi.RemoteException`.
- Because method arguments and returned data will be traveling across the network by means of Java serialization, their data type must be `Serializable`.

You start developing the server side of any distributed Java application by answering the question "What business methods have to be exposed to the client applications and what should their signatures be?" When you know the answer, define remote interfaces that declare those methods.

Listing 25-1 shows the code of the `StockServer` remote interface that will be implemented on the server but must exist on the client side too. This interface declares two business methods: `getQuote()` and `getNasdaqSymbols()`. The first will generate a random price quote for the specified symbol, and the second will return the list of valid stock symbols.



Listing 25-1: StockServer interface

```
package com.practicaljava.lesson25;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface StockServer extends Remote {
  public String getQuote(String symbol) throws RemoteException;
```

```
  public List<String> getNasdaqSymbols()throws RemoteException;
}
```

Implementing Remote Interfaces

Although the remote interface just declares the methods, you need to create a class that will run on the server side and provide implementation for those methods. There is a special requirement to export such a class to the Java RMI run time to enable the class to receive remote calls. This is somewhat similar to binding to a port in the case of `ServerSocket` (see Listing 18-5), but in the case of Java RMI the server also creates a stub — a dummy class (for the client side) that contains proxies of each implemented method from remote interfaces.

The easiest way to export an RMI server instance is by extending it from `java.rmi.server.UnicastRemoteObject`, as in **Listing 25-2**. If your server has to be extended from another class you can explicitly export the server object by calling `UnicastRemoteObject.export()`.

Listing 25-2 shows an implementation of the class `StockServerImpl`, which will process the client's requests. This class generates random price quotes for the stocks located in `ArrayList nasdaqSymbols`.



Listing 25-2: StockServerImpl class

```
package com.practicaljava.lesson25;

import java.rmi.*;
import java.rmi.server.*;
import java.util.ArrayList;

public class StockServerImpl extends UnicastRemoteObject  implements StockServer {
  private String price=null;
  private ArrayList<String> nasdaqSymbols = new ArrayList<String>();

  public StockServerImpl() throws RemoteException {
```

```
    super();

    // Define some hard-coded NASDAQ symbols
    nasdaqSymbols.add("AAPL");
    nasdaqSymbols.add("MSFT");
    nasdaqSymbols.add("YHOO");
    nasdaqSymbols.add("AMZN");
    nasdaqSymbols.add("MOT");
  }

  public String getQuote(String symbol)
                        throws RemoteException {

    if(nasdaqSymbols.indexOf(symbol.toUpperCase()) != -1) {

        // Generate a random price for valid symbols
        price = (new Double(Math.random()*100)).toString();
    }
    return price;
  }

  public ArrayList<String> getNasdaqSymbols()throws RemoteException {
    return nasdaqSymbols;
  }
}
```

Registering Remote Objects

To make a remote object available to clients, you need to bind it to some name in a registry, a naming service that knows where exactly in the network your RMI server `StockServerImpl` is running. This will allow Java clients to look up the object on the host machine by name.

Listing 25-3 depicts the code that binds the instance of `StockServerImpl` to port 1099 on the host machine, which is the local computer in my example. To the rest of the world this server will be known as `QuoteService`.



Listing 25-3: Starting the server and binding it to a registry

```
package com.practicaljava.lesson25;
```

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.net.MalformedURLException;
import java.rmi.RemoteException;

public class StartServer {

  public static void main (String args[]) {

   try {

    StockServerImpl ssi = new StockServerImpl();
    Naming.rebind("rmi://localhost:1099/QuoteService",ssi);

}catch (MalformedURLException e1){
         System.out.println(e1.getMessage());
   }catch(RemoteException ex) {
         ex.printStackTrace();
   }
  }
}
```

There are two methods in the class `java.rmi.Naming` that can bind an object in the registry. The method `bind()` binds an RMI server to a name. It throws `AlreadyBoundException` if the binding already exists. The method `rebind()` replaces any preexisting binding with the new one. In addition to binding a server to a name, this also ensures that the clients requesting such services as `getQuotes()` or `getNasdaqSymbols()` receive their stubs — the local proxies of the remote methods.

The registry must be up and running by the time you start the program in **Listing 25-3**. One way to start the registry is by entering `start rmiregistry` in the Windows command window or `rmiregistry` in Mac OS.

Instead of starting the registry manually, you can start it from within the StartServer program itself by calling the following method:

```
LocateRegistry.createRegistry(1099);
```

If you know that another process has already pre-created the registry, just get its reference and bind the server to it. The `getRegistry()` method can be called without arguments if the RMI registry runs on the default port 1099. If this is not the case, specify the port number (5048 in the following example). The variable `registry` in the following code fragment is a stub to the remote object `StockServerImpl`:

```
StockServerImpl ssi = new StockServerImpl();
Registry registry = LocateRegistry.getRegistry(5048);
registry.bind("QuoteService", ssi);
```

Writing RMI Clients

The client program, running anywhere on the Internet, performs a lookup in the registry on the host machine (using the host machine's domain name or IP address) and obtains a reference to the remote object. **Listing 25-4** shows a sample client program. Notice the casting to the `StockServer` type of the data returned by the method `lookup()`.

Even though the class `StockServerImpl` has been bound to the name `QuoteService`, because this class implements the `StockServer` interface we can cast the returned object to it. The variable `myServer` will see only the methods defined in this interface, while the class `StockSertverImpl` may have other public methods too.



Listing 25-4: RMI client

```
package client;

import java.rmi.*;
import com.practicaljava.lesson25.StockServer;

public class Client {
```

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

   if (args.length == 0) {
     System.out.println("\n Sample usage: java client.Client AAPL");
     System.exit(0);
   }

    try {
        StockServer myServer = (StockServer)
        Naming.lookup("rmi://localhost:1099/QuoteService");

      String price = myServer.getQuote(args[0]);
       if  (price != null){
         System.out.println("The price of " + args[0] +
                            " is: $" + price);
       }
       else{
          System.out.println("Invalid Nasdaq symbol. " +
             "Please use one of these:" +
             myServer.getNasdaqSymbols().toString());
      }

 } catch (MalformedURLException ex) {
       System.out.println(ex.getMessage());
    }catch (RemoteException ex2) {
       System.out.println(ex2.getMessage());
    }
  }
}
```

Security Considerations

Can any RMI client restrict the actions that remotely loaded code can perform on the local computer? Can the server restrict access? You can specify a security policy file containing access restrictions. For example, in the code in Listings 25-3 and 25-4 you can start the `main()` method with the following:

```
if (System.getSecurityManager() == null) {
       System.setSecurityManager(new  RMISecurityManager());
 }
```

The class `java.rmi.RMISecurityManager` extends the class `java.lang.SecurityManager` and provides a security context under which the RMI application executes. In RMI clients the goal is to prevent remotely loaded stub code from downloading unsecured code via remote method invocation.

The RMI client uses a file in which security policies are defined. You can use the default security file, `java.policy`, located in your JDK or JRE installation directory under `lib/security`. The default policy file gives all permissions to the code, but you can create your own file and supply it either via the command-line parameter or in the code before the security manager is set:

```
System.setProperty("java.security.policy", "mypolicyfile");
```

For a more detailed description of security policy files refer to the documentation at **http://download.oracle.com/javase/1.3/docs/guide/security/PolicyFiles.html**.

Java applets can also serve as RMI clients, but they don't need RMI security managers. The only restriction on them is that they can connect only to the RMI server running on the same host on which they were deployed.

Finding Remote Objects

RMI clients find remote services by using a **naming** or **directory service**. A naming service runs on a known host and port number. The subject of naming and directory services will be covered in more detail in Lesson 31.

By now you know that an RMI server can start its own registry that offers naming services for RMI clients. The behavior of the registry is defined by the interface `java.rmi.registry.Registry`, and you saw an example of binding to the registry in the section "Registering Remote Objects."

By default the RMI registry runs on port 1099, unless another port number is specified. When the client wants to invoke methods on a remote object it obtains a reference to that object by looking up the name. The lookup returns to the client a remote reference, aka **stub**.

The method `lookup()` takes the object name's URL as an argument in the following format:

```
rmi://<host_name>[:<name_service_port>]/<service_name>
```

`host_name` stands for the name of a computer on the local area network (LAN), or the name of a domain name system (DNS) on the Internet. `name_service_port` has to be specified only if the naming service is running on a port other than the default. `service_name` stands for the name of the remote object that should be bound to the registry.

Figure 25-1

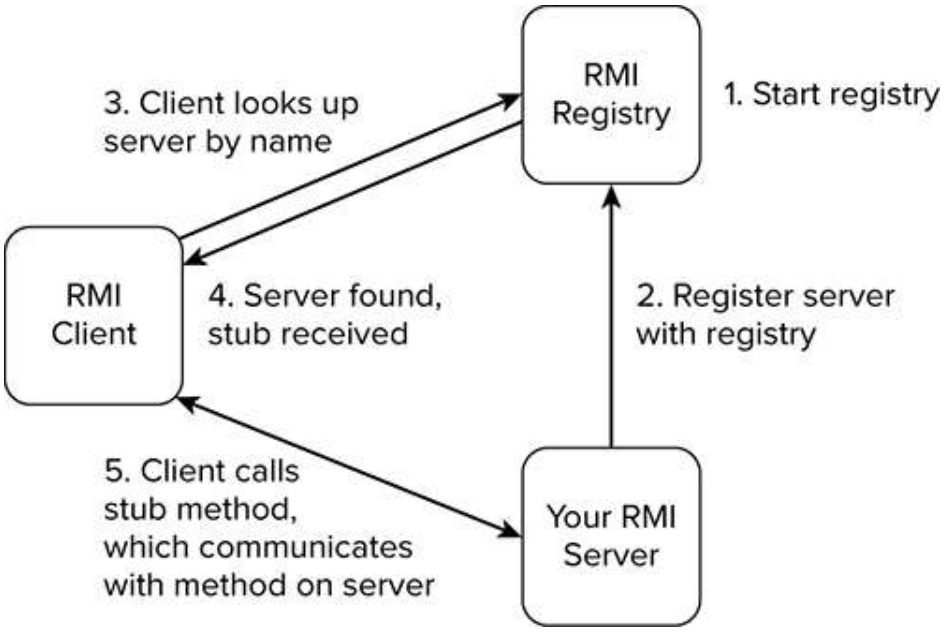


Figure 25-1 illustrates the architecture of an RMI application. In the Try It section you'll implement this architecture for our sample stock quote service.

Try It

In this exercise your goal is to start and test all the parts of the distributed Stock Server application, and you'll run all these parts on the same computer. To emulate multiple computers you'll open three command windows: one for the RMI client, one for the registry, and one for the server. If everything is done properly you should be able to start the RMI client with one of the stock symbols known to the server, and get a price quote back.

Lesson Requirements

You should have Java installed.



You can download the code and resources for this Try It from the book's web page at **www.wrox.com**. You can find them in the Lesson25 folder in the download.

Hints

There is an RMI plug-in for the Eclipse RMI that may be handy for developing RMI-based distributed applications. It contains a useful utility called RMI Spy that shows all outgoing and incoming method calls, and measures execution times. Another useful utility in this plug-in is Registry Inspector, which displays information about objects in the registry. The RMI plug-in is available from **www.genady.net/rmi/index.html**.

Step-by-Step

1. Create an Eclipse project called Lesson25 with two packages: `client` and `com.practicaljava.lesson25`.

2. In `com.practicaljava.lesson25` create the interface `StockServer` that extends `java.rmi.Remote`, as in **Listing 25-1**.

3. In `com.practicaljava.lesson25` create the class `StockServerImpl` that extends `UnicastRemoteObject` and implements the interface `StockServer`, as in **Listing 25-2**.

4. In `com.practicaljava.lesson25` create the class `StartServer` from **Listing 25-3** to start the server and bind it to the naming service.

5. In `client` create the class `Client` that will access the remote server. Follow the code from **Listing 25-4**.

6. Compile all the classes.

7. Open three command windows.

8. Start the RMI registry from the first command window. The naming service will listen to default port 1099. If you are running this program in Windows, use the command `start rmiregistry`. If you are running Mac OS, use the command `rmiregistry`.

Don't expect to see any confirmations that the registry has successfully started. The absence of error messages is your confirmation.

9. In the second command window, start and register `StockServer` with the naming service (registry) that you just started. In the command line specify the system property `java.rmi.server.codebase` — this is where your `StockServer` compiled class is located. I created this example on my MacBook with the user name yfain11, and my Eclipse workspace was located in the directory `practicalJava`. The `codebase` command-line option starts with `-D` and is shown in bold.

```
java -classpath /Users/yfain11/practicalJava/workspace/Lesson25/bin
-Djava.rmi.server.codebase=
file:/Users/yfain11/practicalJava/workspace/Lesson25/bin/ com.practicaljava.lesson25.StartServe
```

After you enter a command similar to the preceding, the server will start and bind to the registry, printing the following message in the command window:

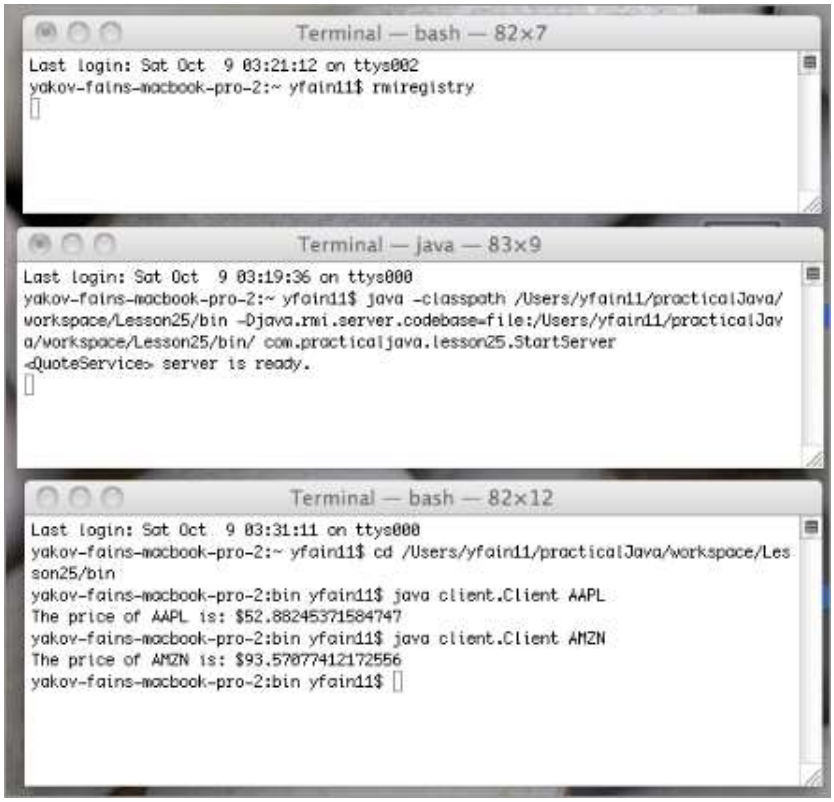
```
<QuoteService> server is ready.
```

10. Open the third command window and switch to the `bin` directory — the `client` sub-directory should be there, as you created it in Step 1. Run the `Client` program, passing the stock symbol as a command line argument, and the `Client` will connect to your "remote" server and receive the price quote, for example:

```
java client.Client AAPL
```

Figure 25-2 shows how the three command windows look on my MacBook. They will look similar in Windows.

Figure 25-2





Please select Lesson 25 on the DVD with the print book, or watch online at www.wrox.com/go/fainjava to view the video that accompanies this lesson.

Chapter 26

Java EE 6 Overview

Starting from this lesson you'll be learning about Java Enterprise Edition (Java EE or formerly J2EE), which is a powerful, mature, and widely used platform for development of distributed applications. The word enterprise here doesn't imply that it is only for large-scale applications. Java EE components are used for the development of everything from an application for a local pizza parlor's website running on a five-hundred-dollar computer to a super-powerful Wall Street trading application that runs on a **cluster** of hundreds of interconnected servers.

This lesson is an overview of the Java EE architecture, concepts, components, and terms that will be covered in detail in the remaining lessons of this book. The goal of these lessons is to give you an understanding of how to approach the development of Java EE applications by showing you how to build small applications rather than making you read the thousand-page manuscript that would otherwise be required for detailed coverage of this subject.

There are an abundance of online materials and books published on the subject of Java EE, and when you will figure out which components are a good fit for your project, finding materials on the selected topic will not be difficult at all. My task is to help you in making that decision and getting you started in the quickest possible way. The Oracle website Java EE at a Glance is a good complement for this lesson: www.oracle.com/technetwork/java/javaee/overview/index.html.

If you start browsing the Internet trying to find more information on Java EE, you can easily get confused by trying to compare the features of different versions of Java EE components. I highly recommend that you stay focused on the features of Java EE 6, which is the most current platform and the easiest to use. It was released in December 2009.

The Big Picture

Go to your favorite online job search engine and search for job listings with the keyword J2EE. Even though J2EE was re-branded into Java EE many years ago, recruiters and job agencies keep referring to this platform as J2EE. Here's one of many job descriptions I found on a major Internet job board:

- **Title**: Java/J2EE Engineer with Core Java, Hibernate
- **Skills**: J2EE, JMS, Spring, MQ, EJB, J2EE, Servlets, JDK, Core Java, Websphere, Junit, log4j, Apache, Oracle, SQL
- The candidate should know:
- * Core Java 6
- * Expertise in EJB, J2EE, Servlets
- * Expertise in Web Services, JAX-WS, JAX-RPC
- * Working knowledge of Hibernate, JDBC, JPA (Oracle)
- * Java Frameworks: JUnit, Log4j
- * Oracle 11, SQL
- * Expertise in Development IDEs: Eclipse/RAD
- * Knowledge of Spring framework
- * WebSphere Application Server v6/7
- * JMS and WebSphere MQ v6/7

How many items from this list of skills did you recognize after mastering the first 25 lessons of this book? You know Core Java 6, JDBC, Eclipse, and SQL. After studying the remaining lessons you'll understand what other skills are required in this ad, why they are required, and how they all fit together. You'll also have technical knowledge of many of the buzzwords listed there.

JCP, JSR, and Other Acronyms

The Java community is accustomed to lots of acronyms. Initially these acronyms might sound intimidating and confusing, but with a little effort they will make perfect sense and explain the way the Java ecosystem lives and functions.

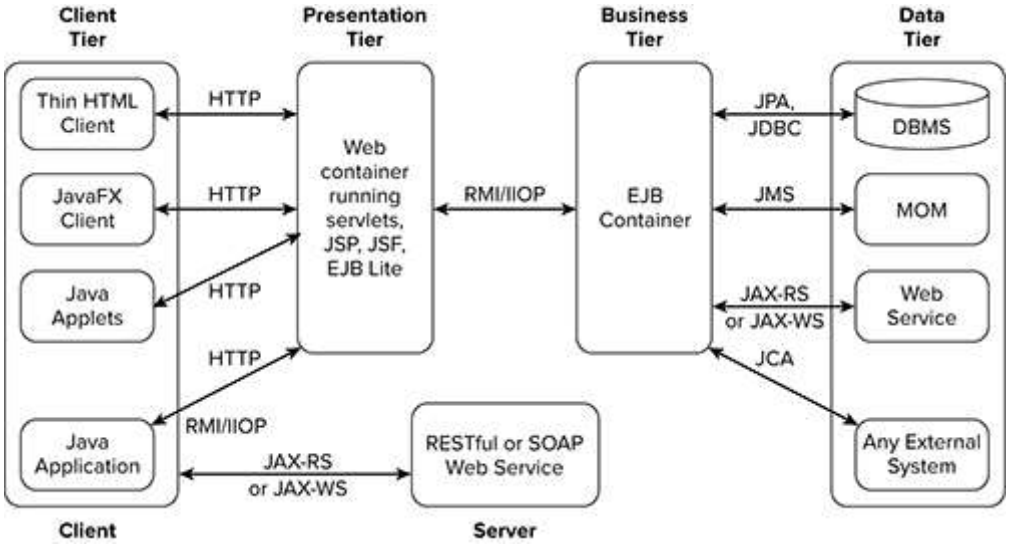
Each version of Java EE includes a set of specifications for various technologies, such as Servlets, JavaServer Pages (JSP), Enterprise Java Beans (EJB), and Java Messaging Service (JMS). Each of these specifications has been defined by an organization called the Java Community Process (JCP). If a person or a group of people decides to propose a specification for some future technology, it will create a Java Specification Request (JSR) and form a group of experts to work on this specification. JSRs are numbered. For example, the specification for Servlets 3.0 was described in JSR 315.

If you decide to get familiar with any specific JSR, visit the website **http://jcp.org**. Currently Java EE includes 45 JSRs, Java SE has 44, and Java ME (Micro Edition covers small devices) has 85. In other words, Java EE is based on standards. If you'd like to get hold of the 33 specifications that were included in Java EE 6, they are described in JSR 316 and are available at **http://jcp.org/aboutJava/communityprocess/final/jsr316/index.html**.

Tiers of Java EE Applications

A typical distributed Java application can be divided into three or four logical tiers. (If the application is getting data directly from a DBMS, as described in Lesson 22, it has a two-tier client-server architecture, and Java EE components are not needed.) **Figure 26-1** shows selected technologies and possible ways of building a distributed application that includes Java EE tiers or layers.

Figure 26-1



The client tier can be implemented on a user's desktop, notebook, or mobile phone, or any other device that has JRE or can connect with Java on the web server. You can create a client as an independent Java application, an applet, or a **thin client** (an HTML/JavaScript file running in a web browser). The word thin refers to the fact that no business processing is being done on the client side, so the server has to work hard processing all application logic for all clients.

If you're building a web application, the web tier (aka the **presentation tier**) comes into the picture. You have your choice of JSP, JavaServer Face (JSF), Servlets, or web services. These components are responsible for the look of your thin client. Java Swing and JavaFX are typically (but not necessarily) used as **fat clients** of web applications. The word fat here refers to the fact that the client can contain some business logic, which minimizes the load on the server.

The business logic of your application is placed in the business tier, which is represented by EJBs. In the past EJBs were pretty heavy components and third-party frameworks such as Spring (see Lesson 35) became a popular choice for implementing the business tier. But Java EE 6 will regain its market share in this department because it's a light POJO- and annotation-based technology that incorporates the best features of the third-party framework. POJO stands for plain old Java object — a Java class that can be written any way you want and doesn't

have to extend any framework class or implement mandatory interfaces. This term is easily understood by people who witnessed the evolution of EJBs, which were Java classes that had to be written and deployed in a certain convoluted way and accompanied by heavy XML configuration files. Things changed drastically, and now EJBs have turned into POJOs marked with annotations. If for some reason you skipped Lesson 24 on annotations, you should go back and study it; otherwise you won't understand most of the remaining material in this book.

While drawing the diagram for **Figure 26-1**, I moved the Web Services box a couple of times between the presentation and business tiers. On one hand, Web Services (see Lesson 34) are based on web communication protocols; on the other hand they can serve as a façade hiding anything, including an entire legacy application written in COBOL on mainframes. In the final version of my diagram, Web Services span both tiers and exist as a separate box in the data tier.

Using DBMSes remains the most popular but not the only way to store data in enterprise Java EE applications. The data can be served by an external web service or arrive as a real-time feed from some messaging infrastructure. MOM stands for message-oriented middleware, and you'll learn what it is in Lesson 30.

Without some practical example, all these new buzzwords may not make much sense to you. You'll see examples in the upcoming lessons, but let's just briefly discuss how you can rebuild the Stock Market application that you already started developed using sockets and RMI.

- Have an applet to connect to `StockQuoteServlet`, which creates what is known as a **session EJB** called `StockMarketQuote`, which connects to external stock exchange software and requests a price quote on the specified symbol(s). This session bean has a timer that updates the quotes every second.

- Do as described in the previous bullet, but replace the applet with a thin HTML client.
- The same as before, but replace the Servlet with a JSP.
- The same as before, but replace the JSP with a JSF.
- The same as before, but replace the session bean with a timer with a message-driven bean (MDB) that subscribes via MOM to the external stock exchange application that sends back new prices only when they change.
- The same as before, but add a session bean with business that will process every price quote received by MDB and apply a secret modeling algorithm, and on certain conditions send a message to buy or sell a specific number of shares of the suggested stock to a third-party trading system via a call to a web service.

Is this enough? I can keep going. As you can see, there are many ways to design a distributed Java EE application. This is what Java architects do for a living.

Containers versus Application Servers

Java EE tiers are implemented in **containers**. It's easy to guess that containers were created to contain something. In the Java EE world containers not only contain, but also control the birth, life, and death of Java components. For example, you don't need to write the `new` statement to create a new instance of an EJB; the container will create a pool of them based on configuration parameters. Even on the client side, while reading about applets you might have noticed that applets' life cycles are controlled by the JRE, which can be called an applet container. Basically, a container is an area inside JVM that can support a lifecycle of certain types of Java objects, such as applets, servlet, EJB, etc.

Containers can take care of various functions, and one of them is thread safety. It's great that multiple clients can connect and make requests to the same server, but can you be sure that a thread initiated

by Mary won't interfere with John's thread? An EJB container implements a single-threaded model ensuring that each client's request operates in a dedicated thread. Containers may offer transactional support with Java Transaction API (JTA) and persist data for you with Java Persistence API (JPA).

In the first few lessons I used a blueprint analogy to explain the relationship between Java classes and objects. I'll use this analogy again, but this time to explain the relationship between the Java EE specification and application servers. The Java EE is a specification, and when a release is published, vendors who want to implement it in their software products create application servers that support this specification.

Multiple vendors offer their versions of a Java EE application server. The question is what version of the Java EE specification they support. Currently the application servers GlassFish (Oracle) and JBoss (Red Hat) support Java EE 6. At the time of this writing WebLogic (Oracle) and WebSphere (IBM) are still on Java EE 5, but should offer support of the latest specification soon.

Java EE application servers have to support multiple containers, for example a Servlet container and an EJB container. Some vendors prefer to create products supporting only certain technologies defined in the Java EE specification. For example, Tomcat (Apache), Jetty (Eclipse Foundation), and Resin (Caucho) offer support for selected technologies (such as Servlets and JSP), which makes them suitable for implementing web applications, but if you are planning to create web applications based on one of these products, you'll need to figure out what other tools or frameworks will be needed to support transactions, the business tier, data persistence, and more.

In this book I'll be using GlassFish 3 from Oracle, which is fully compliant with Java EE 6. I have selected the most widely used technologies

in Java EE development today. In particular you'll be learning about the following:

- Java Servlets 3.0
- JSP 2.2
- EJB 3.1
- JPA 2.0
- JTA 1.1
- CDI 1.0
- JMS 1.1
- JAX-RS 1.1

Two of the acronyms in this list have not been spelled out yet in this book. CDI stands for Context and Dependency Injection and JAX-RS is Java API for RESTful Web Services. You may feel overwhelmed with all these terms, but I'll try to explain them in easy-to-understand language.

Profiles and Pruning

Although Java EE 6 offers a full stack of server technologies, most of the real-world applications don't need all of them. In the past, to get Java EE certified, a vendor of an application server had to implement all JSRs that were listed in the Java EE specification. But most of the applications don't use all technologies included in full Java EE specification. This is how that concept of a profile came about. A profile is a preconfigured subset of Java technologies geared toward solving a specific type of application. Currently, besides the Full Profile, there is the Web Profile, which is designed specifically for the development of web applications. Web profiles include Servlets JSF, JSP, CDI, EJB Lite, JPA, JTA, Interceptors, Managed Beans, and Bean validation.

In the future new profiles may be created to address specific needs of developers. For example, chances are that a new profile will be created for developers for small devices. In general, the mechanism of

lightweight profiles should encourage vendors of web containers to expand their support of Java EE by implementing web profile components. For example, creators of the Resin container are focusing on the implementation of a web profile.

Pruning is a way to reduce the size of the Java EE platform. The pruning process starts with marking in JavaDoc the technologies that are candidates for removal. Then, based on the reaction of the Java community, they will be either removed from the future specifications or not. As an example, some of the Java EE components (such as EJB 2.x) will be removed in Java EE 7.

Miscellaneous Concepts

You may say, "I want my RMI-based stock server application back. Why make things so complex?" The reason is that the only instance of `UnicastRemoteObject` in our RMI `StockServer` may need to process thousands of concurrent user requests, and because of this you'll need to write code ensuring that such concurrent processing is thread safe.

Servlet and EJB containers are scalable and they take care of all multithreading issues, enabling you to write application code as if the application were the only user! This alone should be a good reason for using the Java EE stack as opposed to RMI. Without going into a detailed comparison between RMI and Java EE, I'll just add that if you need to deploy your StockServer application on the Web, corporate firewalls won't allow clients that use the JRMP communication protocol required by RMI.

Java EE is a very robust, reliable, and scalable platform and you will appreciate what its container will do for your code. I'll just mention a couple of concepts here.

If you want Java EE containers to help you, help them by providing configuration parameters (annotations) that specify how many in-