10 Event-driven Programming

Overview

This chapter introduces graphics programming and event-handling in the context of data entry for a car rental company. You will see how to use the new swing components in the Sun GUI library, such as buttons, textfields, menus, and sliders. If you did not read Chapter Eight, you may simply ignore the very few references to JApplet and Graphics2D commands. New language features for this chapter are not used in any other chapter.

- Sections 10.1-10.2 discuss JFrames and how to handle the event of the user clicking on the closer icon.
- Sections 10.3-10.5 introduce objects that listen for a click of a button or a data entry and react to it. They are defined by "inner classes" so they can access the frame's instance variables.
- Section 10.6 completes Version 1 of the CarRental software, which uses the Model/View/Controller pattern.
- Sections 10.7-10.11 describe additional components and listeners, including sliders, timers, combo boxes, radio buttons, and menus. In the process we complete Version 2 of the CarRental software. Arrays are not needed until Section 10.9.

10.1 JFrames, Components, And WindowListeners

You have been hired to create some graphical data entry software by the Wysiwyg Car Rental Agency. Their agents take reservations over the phone, enter them into the database while talking, then send the reservations to the appropriate branch at some airport. You have to develop efficient data entry forms that are pleasant to work with.

JFrames

The `javax.swing` package contains the JFrame and JApplet classes. JFrames are used for applications and JApplets are used for graphical parts of a web page. You begin developing an application program with a graphical user interface (**GUI**) by defining a subclass of the JFrame class from the `javax.swing` package. Objects of the **JFrame** class represent rectangular windows on the display with a title bar and a border.

The JFrame may have a `paint` method just like that of a JApplet, if you want to make drawings and you do not add any components to the JFrame. `paint` is called by the operating system when the user minimizes the window and then brings it back. But since a JFrame does not have a browser to create your graphical object, set its size to a particular width and height, and make it visible, you will need to do that yourself. This initializing process should logically be done in a constructor.

The Painter constructor in the upper part of Listing 10.1 (see next page) creates a Painter object, where Painter is a subclass of the JFrame class. The `super` call invokes the JFrame constructor, supplying the title you want at the top of the frame. Then it calls the `addWindowListener` method, which is explained shortly. It calls three methods to set the size of the frame to a particular width and height in pixels (`setSize`), to cause the frame to appear on the monitor (`setVisible`), and to make drawings (`repaint`) using the Graphics object associated with the frame (`getGraphics()` returns this Graphics object). Failure to specify size and visibility can be disconcerting: The default is an invisible frame 0 pixels wide and 0 pixels tall.

Listing 10.1 The Painter class of objects

```
import java.awt.Graphics2D;

public class Painter extends javax.swing.JFrame
{
   public Painter()
   {  super ("Wysiwyg Car Rentals");  // put the title on it
      addWindowListener (new Closer());
      setSize (760, 600);   // 760 pixels wide, 600 pixels tall
      setVisible (true);    // make it visible to the user
      repaint();            // display the picture
   }  //====================


   /** Draw a pattern on the frame's drawing area. */

   public void paint (java.awt.Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      page.drawString ("pattern", 10, 40);
      for (int depth = 40; depth < 580; depth += 5)
      {  page.draw (new java.awt.geom.Line2D.Double (depth,
                    depth, depth + 30, depth));  // horizontal
      }
   }  //====================
}
//#################################################################


class PainterApp
{
   /** Create and initialize the JFrame. */

   public static void main (String[ ] args)
   {  new Painter();
   }  //====================
}
```

If `jif` is a JFrame object, you may get its title with `jif.getTitle()`, and you may change the title with the method call `jif.setTitle(someString)`. You may find out the current width and height by `jif.getWidth()` and `jif.getHeight()`.

The **Component** class is a superclass of both JFrame and JApplet. The `getGraphics`, `setSize`, `getWidth`, `getHeight`, `repaint`, and `paint` methods are inherited from the Component class, so they are available for both JFrames and JApplets. You should explicitly call the `repaint` method initially for JFrames on which you draw (which asks the operating system to update and paint the Component); your browser calls it for JApplets.

The Painter class does not have a main method. That goes in a separate application class, shown in the lower part of Listing 10.1. It does not assign the constructed Painter object to a variable for later use because no other statement uses it later. If you run this program, you can see how a frame-based application works for drawing figures (the logic in this particular `paint` method is interesting but not important; you should totally ignore it if you did not read Chapter Eight). However, we will not use the `paint` method in the car-rental software we are developing, because the `paint` method should not be used for a JFrame or JApplet on which you put any component (such as a button).

The WindowListener Interface

If Listing 10.1 did not have the `addWindowListener` method call, you could not stop execution of the program without using Control-C in the terminal window. On some machines, the program locks up the computer and you must restart it. It will not let the user stop the program by clicking on the closer button (the X mark) in the upper-right corner of the frame. The window may in fact disappear, but the program still keeps running, which uses processor time and possibly locks up your computer. You need a window closer to tidy up.

Compile the Closer class of Listing 10.2 to define what a Closer object is. Now the `addWindowListener` method call makes arrangements for the program to stop when the user clicks the usual X in the upper right of the window -- that click sends a message to the Closer object to execute its `windowClosing` method. You should add a Closer object as the WindowListener for each JFrame object your programs have.

Listing 10.2 The Closer class

```
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

public class Closer implements WindowListener
{
   /** Enable the closer icon to terminate the program. */

   public void windowClosing (WindowEvent ev)
   {  System.exit (0);
   }  //=====================

   public void windowActivated (WindowEvent ev)     { }
   public void windowDeactivated (WindowEvent ev)   { }
   public void windowIconified (WindowEvent ev)     { }
   public void windowDeiconified (WindowEvent ev)   { }
   public void windowOpened (WindowEvent ev)        { }
   public void windowClosed (WindowEvent ev)        { }
}
```

Interfaces are discussed in detail in Chapter Eleven. For this chapter, all you need to know is that "implements" is very much like "extends" except that (a) the Closer class is called an **implementation** of WindowListener rather than a subclass, and (b) an implementation is required to override all of the methods in its interface.

The **WindowListener** interface and the **WindowEvent** class are in the `java.awt.event` package. The documentation for these two class definitions (depending on your Java version) is in the `jdk1.3\docs\api\java\awt\event` folder. You will see that, to be a WindowListener, a class must implement the seven methods specified in Listing 10.2 (iconifying a window means to minimize it, and activating a window means to give it the focus of keyboard events). However, for simple programs you only need to have the `windowClosing` method to exit the program; the other six can be given the do-nothing implementations shown.

Event-handling

The call of `addWindowListener` in the earlier Listing 10.1 attaches a new Closer object to the JFrame object. When the user clicks on the closer icon at the top-right corner of the JFrame (the big X), the JFrame object sends a `windowClosing` message to whatever WindowListener is attached to it, which would be this Closer object. That `windowClosing` method call executes `System.exit(0)` to terminate the program.

The JFrame object sends additional information to the `windowClosing` method in the form of a parameter of the **WindowEvent** class. This particular `windowClosing` logic does not need that object for anything. But you have to have the parameter anyway. Otherwise you are overloading rather than overriding the method in the WindowListener class. And that would mean that the JFrame object would not call your method. Note: A windowClosing method that does not terminate the program should setVisible(false) for the JFrame object that is being closed. Otherwise the window does not disappear.

Polymorphism

Figure 10.1 shows representations of the JFrame object and the Closer object. **Window** and WindowListener are in the standard Sun library, and JFrame is a subclass of Window. In effect, the Window class has an instance variable of type WindowListener named `wilis` (not its real name, but close). Your Closer object is assigned to `wilis`. This is legal only because Closer implements WindowListener. Then the user's click executes a `wilis.windowClosing(x)` statement that is in a Window method.

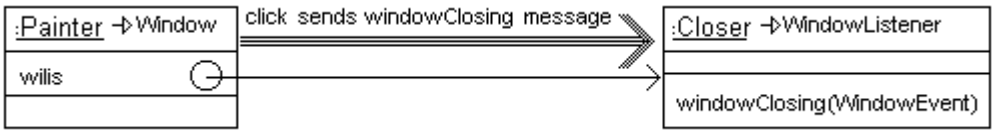

Figure 10.1 The runtime system sends a message to the attached WindowListener

Another program can define e.g. `class Shutter extends WindowListener` and add an object of that type to some JFrame, in which case executing the same `wilis.windowClosing(x)` statement in the Window method could be executing a Shutter method instead of a Closer method. So that statement in the Window method can execute any of several completely different methods (at different times, of course). This is polymorphism.

Exercise 10.1 Rewrite Listing 10.1 to use three more import directives.
Exercise 10.2 Rewrite Listing 10.1 to set the lower-right corner of the window only about ten pixels below and to the right of the drawing it makes.
Exercise 10.3* Read the documentation for WindowListener and then explain the difference between `windowClosing` and `windowClosed`.
Exercise 10.4** Read the documentation for WindowAdapter and then explain the advantage of having a class inherit from it instead of WindowListener, as well as the disadvantage.

10.2 JPanels, Containers, And LayoutManagers

After you talk to the clients for the car rental agency and find out what they want, you decide to make the initial design for the software have the following features (Version 1). Each feature is represented by an object placed on the data-entry form. You will add more features in future versions during the iterative development, such as the time the car will be picked up and the time it will be returned:

- DaysRented: A textfield where the agents enter the number of days for the rental.
- BaseCost: A label giving the cost of a compact-car rental, calculated from the number of days rented and with a reduction for periods of a week or more.
- ClearButton: A button that clears all the entries on the form so a person can start over.
- SubmitButton: A button that writes the entries on the form to the database.
- CustomerName: A textfield for the name of the customer making the reservation.
- CreditCard: A textfield for the customer's 16-digit credit card number.

LayoutManagers

Components are added to Containers. Each Container object has an associated LayoutManager object which determines where components are added to the Container. The simplest LayoutManager is **java.awt.FlowLayout**, which just adds each component in book-reading order: The first component added goes at the top-left; each additional component added goes to the right of the one before, unless there is no room, in which case it goes at the far left of the next row. The components are centered in each row.

The content pane

You cannot add components directly to a JFrame or JApplet object; you must add them to its **content pane**. The `getContentPane()` method for JFrames and JApplets returns a **Container** object. That Container object has a layout manager, a BorderLayout which we will replace by null until we come to Section 10.12. The two key methods you will need from the `java.awt.Container` class are the following:

- `setLayout(someLayoutManager)` replaces the layout manager with a new one.
- `add(someComponent)` adds the given Component as the next item in the list of Components attached to the Container executor. It returns the Component added.

JApplet is a subclass of Component. If you have an applet you want to be part of your frame, your frame has to do what a browser normally does for you: Create the applet, set its size the way you want it, and add it to the content frame. For instance, if Dancer is a subclass of JApplet, the following could be in the constructor of a JFrame:

```
Component degas = new Dancer();
degas.setSize (240, 120);
getContentPane().add (degas);
```

If the applet has a `start` method that a browser would execute automatically, your JFrame has to call that method for the applet as well.

JPanels

The standard way to organize many components on a frame is to use panels. A **JPanel** object is an area to which you can attach various buttons, textfields, etc. You then place the JPanel on the content pane of a JFrame or JApplet. JPanel is a subclass of `javax.swing.JComponent`, which is a subclass of `java.awt.Container`, which is a subclass of `java.awt.Component`.

A JPanel object has a FlowLayout. This car rental software uses the following two methods from the `javax.swing.JPanel` class, as well as the `add` method available for any Container object:

- `new JPanel()` creates a new JPanel object with a FlowLayout.
- `setBounds(xInt, yInt, widthInt, heightInt)` sets the top-left corner of the executor at <`xInt, yInt`>, with the given `widthInt` and `heightInt` measured in pixels. The executor must be a Component whose container has a null layout (which is what we have for the JFrame's content pane).

The calculator of the base cost will be placed on the top panel on the frame, and the customer name and credit card will be entered on the second panel on the frame. Other components will be in three additional panels. Since it is by far best to keep the methods short, we will call five separate methods to obtain the five separate panels, e.g., `subViewOne()` returns a panel with textfields and labels already added.

The coding so far is in Listing 10.3 with all the import directives that will be needed. All parts of that coding have been explained in the preceding few pages. The reason that the name of the class is CarRentalView is that it provides the user's view of this software. The view has five subviews, represented by the five panels on the frame. The five methods that will be developed later are stubbed to make this coding compilable as is.

Listing 10.3 The CarRentalView class, some methods stubbed

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

public class CarRentalView extends JFrame
{
   public CarRentalView()
   {  super ("Wysiwyg Car Rentals");  // put the title on it
      this.addWindowListener (new Closer());  // enable closing
      this.setSize (760, 600);   // cover most of the screen
      this.init();
      this.setVisible (true);    // make it visible to the user
   }  //====================

   public void init()
   {  Container content = this.getContentPane();
      content.setLayout (null);
      content.add (subViewOne());
      content.add (subViewTwo());
      content.add (subViewThree());
      content.add (subViewFour());
      content.add (subViewFive());
   }  //====================

   private JPanel subViewOne()          { return new JPanel(); }
   private JPanel subViewTwo()          { return new JPanel(); }
   private JPanel subViewThree()        { return new JPanel(); }
   private JPanel subViewFour()         { return new JPanel(); }
   private JPanel subViewFive()         { return new JPanel(); }
}
//#################################################################


class CarRentalApp
{
   /** Create and initialize the JFrame. */

   public static void main (String[ ] args)
   {  new CarRentalView();
   }  //====================
}
```

Everything that is needed for JFrames but not for JApplets is kept in the CarRentalView constructor, and everything else is done by the `init` method. So if you wanted to make this software into an applet instead, you would only need to change "JFrame" to "JApplet" in the class heading and comment out the CarRentalView constructor. Your browser will automatically create the JApplet, size it, make it visible, and call its `init` method. The same convertibility was maintained in the earlier Listing 10.1: If you change "JFrame" to "JApplet" and comment out the Painter constructor, you get an applet for which your browser will call the `paint` method.

The overall logic of the `subViewOne` method will be as follows. These statements create a panel, attach a few Components, and return the panel to be attached to the JFrame's content pane. That way, whatever you attach to the panel will be inside the frame. The second statement sets the size of the panel with a width that is slightly less than the entire frame, to allow for the border on the frame:

```
// addFirstPanel, statements 1-3
JPanel panel = new JPanel();
panel.setBounds (10, 25, this.getWidth() - 20, 40);
// statements attaching Components, e.g., panel.add(x);
return panel;
```

You are probably starting to become confused about the various standard library classes you have seen, particularly their methods and relationships. Figure 10.2 summarizes all that you have seen so far, except that some unneeded intermediate classes are left out (specifically, JFrame is a subclass of java.awt.Frame which is a subclass of Window, and JApplet is a subclass of java.applet.Applet which is a subclass of Panel).

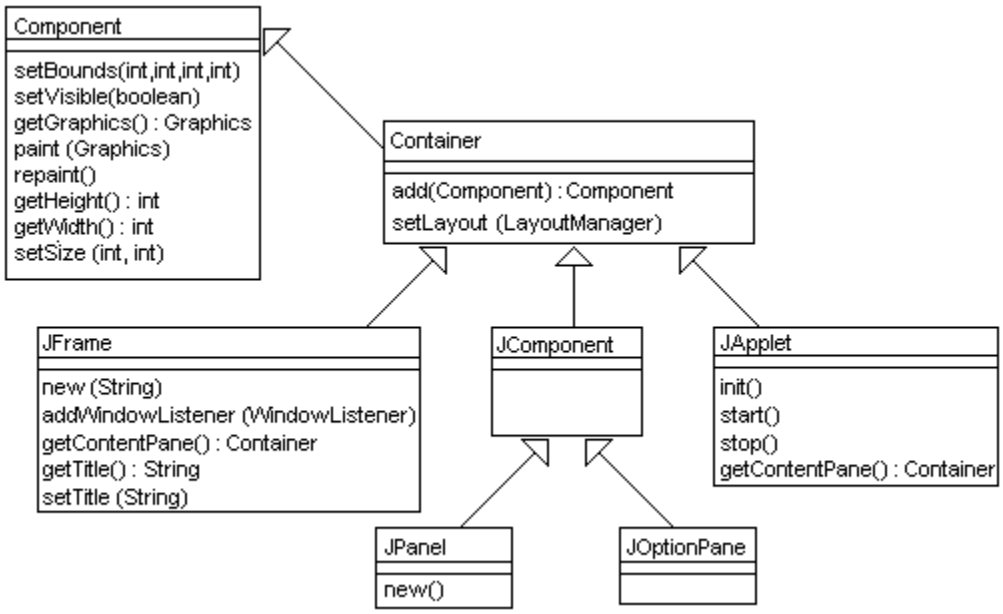

Figure 10.2 Hierarchy for some standard library classes

An optional section at the end of this chapter describes layout managers that are easier to use. For instance, if you replace the second statement of the `init` method in Listing 10.3 by the following, you can omit all statements in CarRentalView that set the bounds of a panel. The layout manager will then re-adjust the height and width of each panel whenever the user changes the shape of the frame:

```
content.setLayout (new BoxLayout (content, BoxLayout.Y_AXIS));
```

You could instead set the layout manager of the content pane as `new FlowLayout()`, which allows you to add many components in book-reading order. However, the `setSize` method for a JPanel you add using FlowLayout may not take effect.

Exercise 10.5* Read the documentation for LayoutManager and then list three kinds of layouts that implement the LayoutManager interface.
Exercise 10.6* Draw the UML class diagram for Listing 10.3.

10.3 JLabels, JTextFields, And ActionListeners

The **JLabel** class provides objects that simply display messages. The user cannot change the message, nor can the user get a reaction from a JLabel by clicking on it. The `javax.swing.JLabel` class is a subclass of JComponent, so you may use a JPanel's `add` method to add it to the panel. JLabel has the following useful methods:

- `new JLabel(textString)` creates a JLabel with the given String as its text.
- `setText(textString)` changes the text on the JLabel to the given String.
- `getText()` returns the String value that is the current text on that JLabel.

Quite often, a statement such as `panel.add (new JLabel ("customer name"));` does everything that has to be done with that particular label. If you do not expect to refer to the label later, it is superfluous to assign the newly created object to a variable. The label is therefore **anonymous**.

JTextFields

A **JTextField** object appears as a rectangular area that the user can fill in. You have surely seen JTextFields on web pages where you entered your name, address, or password. The `javax.swing.JTextField` class has the following methods:

- `new JTextField(textString)` creates a JTextField with the String as its text.
- `new JTextField(widthInt)` creates a JTextField with the specified number of columns. For variable-width fonts, a column width is the pixel width of the letter 'm'.
- `addActionListener(someActionListener)` attaches a **listener** object to the JTextField. The listener object executes a method named `actionPerformed` when the ENTER key is pressed inside the JTextField.

A JTextField is a subclass of **JTextComponent**, which is a subclass of JComponent. The `javax.swing.text.JTextComponent` class has the following methods:

- `setText(textString)` changes the text on the JTextField to the given String.
- `getText()` returns the String value that is the current text on that JTextField.

The subViewOne method

The first panel of the CarRental software is to contain a little calculator that computes the base cost of a car from the number of days it is rented. The formula that the client specifies is the number of days, minus one day for each whole week in the rental period, all multiplied by $30. The actual cost of a rental is the base cost for a compact and some multiple of the base cost for a nicer car. The user will type a number of days into a JTextField instance variable named `itsDaysRented` and press the ENTER key. Then the listener object that reacts to that input will make the corresponding change in the value stored on a JLabel instance variable named `itsBaseCost`.

Figure 10.3 shows how the panel will look on the display (except that the panel itself is invisible, so you will not see the larger rectangular outline on the display). The panel has a JLabel that says "Enter days rented:" and a JLabel that says "The base cost is:". These two labels are not given names in the program, because they are not referred to later in the execution. But the JTextField `itsDaysRented` is given a name when `subViewOne` creates it because the listener object later needs to retrieve its text. And the JLabel `itsBaseCost` is given a name when `subViewOne` creates it because the listener object later needs to change what `itsBaseCost` says.

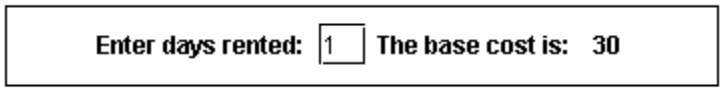

Figure 10.3 Objects on the first panel

Listing 10.4 has the coding required for the `subViewOne` method. The `panel.add` method calls attach the given Components to the panel in the order that they occur. The following statement should be the only puzzler in the `subViewOne` method:

```
itsDaysRented.addActionListener (new DaysRentedAL());
```

Listing 10.4 The subViewOne portion of CarRentalView

```
// CarRentalView class, part 2

   public static final double PRICE_PER_DAY = 30.00;
   ////////////////////////////////
   private JTextField itsDaysRented = new JTextField ("1");
   private JLabel itsBaseCost = new JLabel ("" + PRICE_PER_DAY);


   private JPanel subViewOne()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 25, this.getWidth() - 20, 40);

      panel.add (new JLabel ("Enter days rented:"));
      this.itsDaysRented.addActionListener (new DaysRentedAL());
      panel.add (this.itsDaysRented);
      panel.add (new JLabel ("The base cost is:"));
      panel.add (this.itsBaseCost);

      return panel;
   }  //=====================


   private class DaysRentedAL implements ActionListener
   {
      public void actionPerformed (ActionEvent ev)
      {  int d = Integer.parseInt (itsDaysRented.getText());
         itsBaseCost.setText ("" + (d - d / 7) * PRICE_PER_DAY);
      }
   }  //=====================
```

ActionListeners

The basic listener logic for JTextFields is: `addActionListener` attaches a listener object to the JTextField object. The listener object "listens" for the user to press the ENTER key while the cursor is within the JTextField. When that happens, the listener object executes its `actionPerformed` method, performing whatever actions you have coded as an appropriate response to that keypress. This is precisely analogous to what happens when the user clicks the closer button for a JFrame and a WindowListener object executes its `windowClosing` method.

A listener object is called a **functor** or **function object**, since its only purpose is to supply a function (another name for a method). Functors have instance methods (`actionPerformed` in this case) but usually do not have instance variables.

The parameter of the `addActionListener` method must be an object from a class that implements the **ActionListener interface**. In the lower part of Listing 10.4, it is the DaysRentedAL class ("AL" is for "action listener"; the next section explains why the DaysRentedAL class is declared inside of the CarRentalView class). The ActionListener interface specifies just one method, whose heading is as follows:

```
public void actionPerformed (ActionEvent ev)
```

Event-driven programming

For **event-driven programming**, you **register an event-handler** using an `addActionListener` method or the equivalent. The runtime system waits until a user event or other action causes a JTextField or JButton or other Component to send a message to the listener object registered with it. This makes the runtime system execute the corresponding method, then go back to waiting. It will not react to a new event until the previous action is done. It can react only if it is in this "waiting state."

This is why `System.exit(0)` must be executed at the end of a program that uses graphic components. When a graphic component is created by the program, the runtime system goes into a mode where it remains in the waiting state whenever it has nothing to do. So nothing is happening with the program, but it is still running, waiting for listener actions. Even if you close all windows, the program keeps waiting. It takes a `System.exit` command to terminate the program's demand on system resources.

The JTextField object sends additional information through the `actionPerformed` method in the form of an object of the **ActionEvent** class. This particular `actionPerformed` logic does not need that object for anything. But you have to have the parameter anyway, because the interface requires it.

ActionListener is an interface in the `java.awt.event` package, just as is WindowListener; and ActionEvent is a class in the `java.awt.event` package, just as is WindowEvent. This is the reason for the import directive in the earlier Listing 10.2.

JPasswordFields

JPasswordField is a subclass of JTextField. The `javax.swing.JPasswordField` class has two key methods:

- `new JPasswordField(widthInt)` yields a component that differs from JTextField in that, as the user types, an asterisk appears for each character instead of the character typed. This keeps other people from seeing the password being entered. The int parameter gives the width in characters (using an 'm' to measure it).
- `getPassword()` is like `getText()` except it returns a char array instead of a String, which minimizes security problems. Your coding can go through the characters in the array one at a time to check it against the true password and then erase those characters. If you store the password in a String object using e.g. `getText()`, it will not be erased from memory.

Exercise 10.7 What would be the effect of swapping the third and fifth statements in the `subViewOne` method of Listing 10.4?
Exercise 10.8 Revise Listing 10.4 to have another label at the end of the panel with the phrase "for a compact" on it.
Exercise 10.9 Revise the `actionPerformed` method in Listing 10.4 so it interprets the input value as a double temperature in degrees Fahrenheit and displays on the last label the corresponding temperature in degrees Centigrade.
Exercise 10.10* Read the documentation for JTextField, then write one statement that doubles the width of a JTextField object named `sam`.

10.4 Inner Classes

You probably noticed the problem with the logic in Listing 10.4: How can a method in the DaysRentedAL class refer to an instance variable in the CarRentalView class? It is bad form to make instance variables public unless they are final variables. But only statements inside of a class can mention private instance variables in that class.

The solution is simple: Put the DaysRentedAL class inside of the CarRentalView class. Now it can mention those variables. Java allows you to define one class inside another, which makes it an **inner class**. The class it is inside of is called its **outer class**. An inner class can have instance variables and methods like the classes you are used to. It cannot contain class methods or class variables, since its heading does not include the `static` modifier. An inner class can mention any method or field variable in its outer class. Also, the outer class can mention any public method or field variable declared within an inner class (though this is not done in this chapter).

In summary: An inner class member of a larger class Out looks like a regular class except (a) it can refer to members of the Out class, and (b) we normally replace `public class` by `private class` in the class heading to keep any outside class from mentioning DaysRentedAL.

The secondary default executor

The `new DaysRentalAL()` command is only allowed inside an instance method or constructor in the outer class. It creates a DaysRentalAL object that "knows" its creator, which is the executor (`this`) of that CarRentalView instance method. Inside a DaysRentalAL method, you can refer to that creator as **CarRentalView.this**. So the two mentions of CarRentalView instance variables inside the `actionPerformed` method could be written as follows:

```
CarRentalView.this.itsDaysRented.getText()
CarRentalView.this.itsBaseCost.setText (whatever)
```

However, the compiler allows you to omit the `CarRentalView.this` specification. As you know, the compiler assumes that any use of an instance variable or method that has no stated executor has `this` for its executor. But if `this` does not have a variable or method of that name, then the compiler assumes it has `Out.this` as the executor, where Out stands for the outer class. In other words, the default executor is the inner class's object if applicable, otherwise it is the outer class's object. Since the DaysRentedAL class does not have field variables with the names `itsDaysRented` and `itsBaseCost`, the compiler correctly interprets the mentions of those variables as mentions of the CarRentalView object's instance variables. In short, `CarRentalView.this` is the **secondary default executor**.

Every JTextField has an instance variable (say it is called `alis`) that refers to an ActionListener kind of object. When the ENTER key is pressed within the JTextField, it creates an event `ev` and executes `alis.actionPerformed(ev)`.

Figure 10.4 shows representations of some of the various objects involved in this program. A JFrame has in effect a Container instance variable (it might be named `contentPane`) that refers to its content pane. That pane has an instance variable (it might be named `comp`) that is an array of components that have been added to the pane. One of those components is the first panel added to the content pane. That first panel also has an array of components that have been added, and the second one of them is the `itsDaysRented` JTextField (because the first one is an anonymous JLabel).

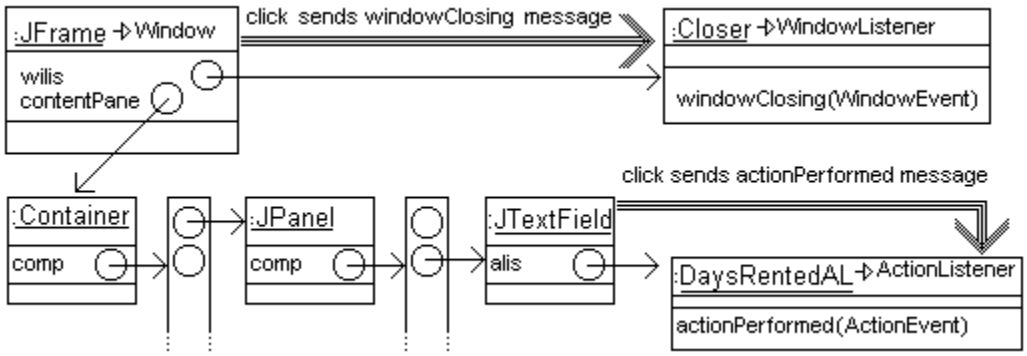

Figure 10.4 The runtime system sends messages to listeners

Technical Note If X is an inner class of Out, `someOutObject.new X(whatever)` is a call of a constructor of class X. That is, `new` for an inner class requires an executor of class Out. When you use `new` alone inside an instance method or constructor of Out, you implicitly call `this.new X(whatever)`. In particular, the fourth statement of the `subViewOne` method could be written as follows:

```
itsDaysRented.addActionListener (this.new DaysRentedAL());
```

Language elements
One class can be declared inside of another class with heading: public class ClassName
or: private class ClassName
The inner class cannot have class variables or class methods, but it can have constructors.
Only an instance x of the outer class can call the constructor, e.g.: x.new ClassName (parameters).
Within an instance method of the inner class, Out.this refers to the instance of the outer class Out that called the constructor to create the instance of the inner class.

Exercise 10.11 (harder) What changes would be required in Listing 10.4 to have SubViewOne be a private inner class of CarRentalView and also a subclass of JPanel, so that its constructor would be called from the `init` method in Listing 10.3 by the statement `content.add (new SubViewOne())` but still have the same effect?
Exercise 10.12 (harder) Explain what changes would be needed to have the DaysRentedAL class outside of the CarRentalView class, not an inner class, and still have it do its job.
Exercise 10.13* Revise Listing 10.4 so that it has another label and textfield saying "Senior discount =" followed by "0". Each time the user clicks the ENTER key in either this new textfield or in `itsDaysRented`, the value in `itsBaseCost` is updated to be X dollars less than it is calculated to be in Listing 10.4, where X is the current dollar value in the new textfield.
Exercise 10.14* Read the documentation for JTextField and find the true name of the variable that stores the ActionListener object (it is not `alis`).

10.5 JButtons And EventObjects

A JButton is an object you can attach to any Container object. You usually attach an ActionListener object to a JButton. When the user clicks on the button, that sends a message to whichever ActionListener kind of object is attached to the button. That ActionListener kind of object executes its `actionPerformed` method, thus performing whatever actions you specify as the proper response to the button click.

The following methods are available for use with JButtons. **JButton** is a subclass of the **AbstractButton** class, from which it inherits all of these methods except the constructor. Both of these classes are in the `javax.swing` package:

- `new JButton(textString)` constructs the JButton object and writes the given string of characters on the button.
- `getText()` retrieves the string of characters written on the button.
- `setText(textString)` changes the string of characters written on the button. For instance, you can change the words on a button named `butn` to be "press me" with the statement `butn.setText("press me")`.
- `setMnemonic(someChar)` establishes a hotkey. So `butn.setMnemonic('c')` makes it so that a user who presses the letter 'c' while holding down the ALT key gets the same effect as clicking on the button named `butn`. The letter 'c' is underlined in the name on the button.
- `addActionListener(someActionListener)` is the same as for JTextField.

Example of using a button and an ActionListener

By way of illustration, you could have the following class for the kind of listener object that displays a message each time a button is clicked:

```
public class Ear implements ActionListener
{
   public void actionPerformed (ActionEvent ev)
   {  JOptionPane.showMessageDialog (null, "don't do that!");
   }  //======================
}
```

Now a method in a subclass of JFrame could have the following statements to create a JButton with the word "squawk" on it and attach the button to the frame's content pane. Clicking the button sends the `actionPerformed` message to the Ear that is attached to the JButton and thus displays the message "don't do that!" You could add the same Ear object to each of several buttons as their ActionListener, so that all would produce that same dialog box:

```
JButton butn = new JButton ("squawk");
butn.addActionListener (new Ear());
this.getContentPane().add (butn);
```

ActionEvent is a subclass of **EventObject**, which has a `getSource()` method that returns the Object value that produced the event. By way of illustration, if an Ear object is attached to several different JButtons, you could have it add an exclamation point to the text displayed on whatever button produced the event by putting the following two statements in the `actionPerformed` method of the Ear class. The returned value of `ev.getSource()` has to be cast to the JButton subclass, since `getSource` returns a generic Object value:

```
JButton source = (JButton) ev.getSource();
source.setText (source.getText() + "!");
```

A TicTacToe example

As one more illustration, a TicTacToe applet might have nine buttons, each initially with the word "open" on it. A player clicks an open button to change it to "X", and the computer opponent then clicks an open button to change it to "O". So the one `actionPerformed` method for all nine buttons could be as follows, assuming that the ActionListener implementation has access to an instance variable named `itsGame` which refers to an object with an appropriate `checkForGameOver` method:

```
public void actionPerformed (ActionEvent ev) // for TicTacToe
{  JButton source = (JButton) ev.getSource();
   if ( ! source.getText().equals ("open"))
      JOptionPane.showMessageDialog (null, "Already taken");
   else
   {  source.setText ("X");
      itsGame.checkForGameOver (source);
   }
}  //======================
```

You can draw on a JButton or JLabel object, or put a border on it, if you declare a subclass of JButton or JLabel with a method that overrides the standard `public void paint(Graphics g)` method. The drawing instructions go inside that method.

The subViewTwo method

The `subViewTwo` method in the CarRentalView class has the following statements to create a button with the word "Clear" on it and attach an ActionListener instance of the ClearButtonAL class. Clicking this clear button sends the `actionPerformed` message to that listener. The button is added to a JPanel named `panel` which is attached to the content pane of the frame:

```
// statements in subViewTwo
JButton clearButton = new JButton ("Clear");
clearButton.setMnemonic ('c');
clearButton.addActionListener (new ClearButtonAL());
panel.add (clearButton);
```

The action that a ClearButtonAL is to perform when notified is to clear out the values written on the two JTextFields named `itsCustomerName` and `itsCreditCard`. So the following private inner class is put inside the CarRentalView class, to allow the `actionPerformed` method to access the two private instance variables:

```
private class ClearButtonAL implements ActionListener
{
   public void actionPerformed (ActionEvent ev)
   {  CarRentalView.this.itsCustomerName.setText("");
      CarRentalView.this.itsCreditCard.setText("");
   }
}  //======================
```

A drawing of the objects and messages involved here would be almost the same as Figure 10.4: The second component of the content pane's array of components is the second panel. The first component of that panel's array of components is the clear button, which has a ClearButtonAL object to which an `actionPerformed` message is sent when the button is clicked.

The second panel also needs a submit button that the user clicks when the data recorded on the screen is to be stored in the customer database. The click will send an `actionPerformed` message to a SubmitButtonAL object.

A summary example

The next listing illustrates an easy way to use a frame for a GUI implementation of a program. It uses a label, a button, and a text field, as well as the text area and scroll pane objects described in Section 6.7. The user clicks the message dialog's OK button to make the frame go away and thus terminate the program. Remember that you may have any displayable object in place of the String value for `showMessageDialog`.

Listing 10.5 implements a queue of characters -- you may add characters to the rear or take them off the front. Specifically, each time that the user presses the ENTER key in the text field, that adds characters from the text field to the rear of the data; and when the user clicks the button, that removes the first character from the front of the data. The text area has scrollbars; it displays the state of the data after each change.

Listing 10.5 A complete program illustrating basic components

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

public class Gooey extends JPanel
{
   private JTextField itsInput = new JTextField (15);
   private JTextArea itsOutput;
   private String itsData = "";


   public Gooey (int rows, int columns)
   {  this.add (new JLabel ("entry:"));
      itsInput.addActionListener (new TextFieldAL());
      this.add (itsInput);
      JButton button = new JButton ("delete first character");
      button.addActionListener (new ButtonAL());
      this.add (button);
      itsOutput = new JTextArea (rows, columns);
      this.add (new JScrollPane (itsOutput));
   }  //======================


   private class TextFieldAL implements ActionListener
   {  public void actionPerformed (ActionEvent ev)
      {  itsData += itsInput.getText();
         itsInput.setText ("");
         itsOutput.append (itsData + "\n");
      }
   }  //======================


   private class ButtonAL implements ActionListener
   {  public void actionPerformed (ActionEvent ev)
      {  if ( ! itsData.equals (""))
            itsData = itsData.substring (1);  // delete the first
         itsOutput.append (itsData + "\n");
      }
   }  //======================
}
//#################################################################


class GooeyApp
{
   public static void main (String[] args)
   {  JOptionPane.showMessageDialog (null, new Gooey (8, 20));
      System.exit (0);
   }  //======================
}
```

Exercise 10.15 Revise the `actionPerformed` method in the original Ear class to change the words on the button that was clicked to "Ouch!".
Exercise 10.16 Revise the `actionPerformed` method for TicTacToe to print two different error messages depending on which of the two illegal choices the user made, a button with "X" on it or a button with "O" on it.
Exercise 10.17* Revise the `actionPerformed` method in the ClearButtonAL class so that it asks the user "Are you sure?" using a JOptionPane method and does not make any change if the user does not confirm it.

Part B Enrichment And Reinforcement

10.6 The Model/View/Controller Pattern

The CarRental software is an example of the Model/Controller/View pattern, which is illustrated at the end of Chapter Six for the RepairShop software. Specifically, the CarRentalView subclass of JFrame displays the current values on the monitor for the user to see, i.e., presents the View into the program. The View has five subViews, represented by the five panels added to the content pane of the frame. Another part of the software maintains the customer database but normally does no input or output; it is the data Model for this software. The third part of the software is the Controller; it accepts input and sends messages to the Model and the View to update themselves accordingly. With the two panels added so far, we have three controller objects: DaysRentedAL, ClearButtonAL, and SubmitButtonAL.

The preceding Listing 10.5 provides another example of the Model/View/Controller pattern: `itsData` is the model, the two `actionPerformed` methods are for the controller objects, and the rest of the Gooey class is the view.

We need some kind of object in which to store information about customers. This object is the Model part of the Model/View/Controller pattern. For the initial versions of the software, we will have a prototype CarRentalModel class, barely adequate to let us test the user interface. The following class will suffice for now:

```
public class CarRentalModel
{  public void add (String given)
   {  System.out.println ("Model got " + given);
   }  //======================
}
```

The completed `subViewTwo` method is in Listing 10.6 (see next page), along with the relevant instance variables and ActionListener implementations (the ClearButtonAL class illustrates how the secondary default executor is explicitly referenced).

Note that it is quite legal to declare instance variables wherever you want in the class. There is a lot to be said for declaring them only immediately before the methods that use them when you have a class of many pages. Figure 10.5 shows what the display looks like during execution of this program so far (Version 1 in the iterative development).

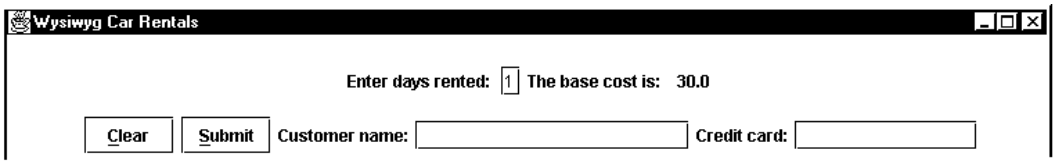

Figure 10.5 Screen shot of execution of CarRentalApp, Version 1

Listing 10.6 The subViewTwo portion of the CarRentalView class

```
// CarRentalView class, part 3

   private CarRentalModel itsModel = new CarRentalModel();
   private JTextField itsCustomerName = new JTextField (18);
   private JTextField itsCreditCard = new JTextField (12);


   private JPanel subViewTwo()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 75, this.getWidth() - 20, 40);

      JButton clearButton = new JButton ("Clear");
      clearButton.setMnemonic ('c');
      clearButton.addActionListener (new ClearButtonAL());
      panel.add (clearButton);

      JButton submitButton = new JButton ("Submit");
      submitButton.setMnemonic ('s');
      submitButton.addActionListener (new SubmitButtonAL());
      panel.add (submitButton);

      panel.add (new JLabel ("Customer name:"));
      panel.add (this.itsCustomerName);
      panel.add (new JLabel ("Credit card:"));
      panel.add (this.itsCreditCard);

      return panel;
   }  //=====================

   private class ClearButtonAL implements ActionListener
   {
      public void actionPerformed (ActionEvent ev)
      {  CarRentalView.this.itsCustomerName.setText("");
         CarRentalView.this.itsCreditCard.setText("");
      }
   }  //=====================

   private class SubmitButtonAL implements ActionListener
   {
      public void actionPerformed (ActionEvent ev)
      {  itsModel.add (itsCustomerName.getText() + "  "
                     + itsCreditCard.getText());
      }
   }  //=====================
```

The JComponent class

The `setToolTipText(String)` method call sets the text that will appear when the mouse cursor pauses over a component. This method is in the JComponent class from the `javax.swing` package. JButtons, JTextFields, and JLabels are all subclasses of the **JComponent** class, a subclass of the Container class. So you can add a tool-tip to any of those swing components. For instance, you could put the following two statements in the `subViewTwo` method of Listing 10.6, one after each JButton constructor call:

```
   clearButton.setToolTipText ("clear all entries on the form");
   submitButton.setToolTipText ("store all values in the file");
```

Figure 10.6 shows the relationships of the additional subclasses of JComponent discussed in the past few sections, plus the JSlider class to be discussed in the next section. The figure includes all of the methods used in this book for these classes. There are many more swing classes in the standard library; JComponent alone has more direct subclasses than Bach had children (Maria gave him seven and Anna gave him thirteen). The constructors are not mentioned in this figure for the AbstractButton and JTextComponent classes. That is because they are abstract classes (described in Chapter Eleven), which means you cannot create objects of those classes.

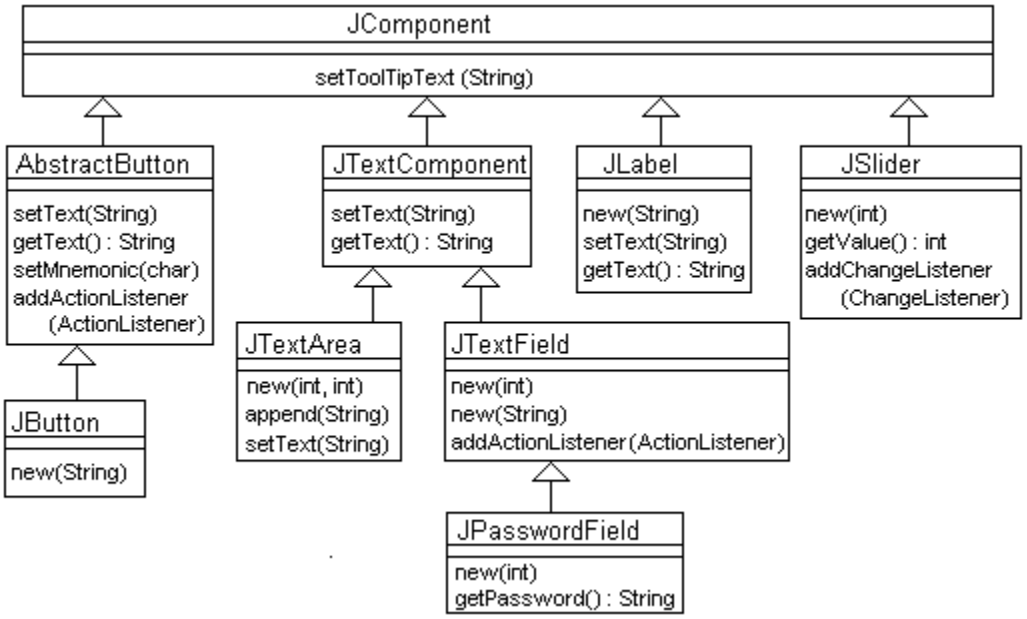

Figure 10.6 UML class diagram of subclasses of JComponent so far

Anonymous classes

Java allows great flexibility in making listener objects. It may seem natural to have each button listen for its own click, but there are times when you want to have a single ActionListener object for several different buttons or other objects.

If you want an ActionListener object to only listen for one Component, you may write the specification for an action listener more compactly as an **anonymous class**. That is a class without a name, for which you only supply the body without constructors. For instance, you may omit the ClearButtonAL class completely from Listing 10.6 if you replace the fifth statement of the `subViewTwo` method by the following:

```
clearButton.addActionListener (new ActionListener()
      {  public void actionPerformed (ActionEvent ev)
         {  itsCustomerName.setText ("");
            itsCreditCard.setText ("");
         }
      }
);  // this parenthesis matches the one before "new"
```

All this does is create a single object of the anonymous class, which is an implementor of the ActionListener class and has the `actionPerformed` method specified. You can use this kind of construction, where the body of anonymous class X immediately follows the parentheses of a constructor call `new Whatever()`, wherever you need just one new instance of class X and (a) X is a subclass of class Whatever, or (b) X implements interface Whatever.

An anonymous class is an inner class. In general, it is best to avoid anonymous classes unless they are extremely short, rarely over four or five statements in them. This book does not use anonymous classes anywhere outside of this section.

Exercise 10.18 Where would the textfield for the credit card numbers appear if the panel were not wide enough to have it after the other components?
Exercise 10.19 What changes would be needed in Listing 10.6 to have the two buttons moved to the right of the credit card number?
Exercise 10.20 (harder) Revise the second `actionPerformed` method in Listing 10.6 so it prints an error message if either the customer name or the credit card is the empty string.
Exercise 10.21* Revise Listing 10.6 to have two textfields for the customer's name (first name and last name) instead of just one.
Exercise 10.22* Rewrite Listing 10.6 to add either "this" or "CarRentalView.this" wherever possible.
Exercise 10.23* How would you rewrite Listing 10.6 to have SubmitButtonAL be an anonymous class?
Exercise 10.24* Read the documentation for JComponent, then describe three advantages of the new swing components over the original ones (Button, TextField, etc.).
Exercise 10.25** Draw the UML diagram for the entire CarRentalView program so far.

10.7 JSliders, JTextAreas, And ChangeListeners

When you have several lines of output, a JTextField or JLabel does not work well to display the output. A **JTextArea** is better, since it is a rectangular area on the drawing surface. The `javax.swing.JTextArea` class is a subclass of JTextComponent, so you can use the `setToolTipText`, `setText`, and `getText` methods with JTextAreas. The methods you saw in Chapter Six are all you really need for JTextAreas:

- `new JTextArea(rowInt, columnInt)` creates a JTextArea with the given number of rows and columns of characters. The width is calculated from the width of the letter 'm'.
- `setText(textString)` puts the given String value in the JTextArea, replacing whatever was already there. In particular, `answer.setText("")` erases everything in the JTextArea named `answer`.
- `append(someString)` adds some more information to whatever is already in the JTextArea. To start a new line with the information you add, use the command `answers.append('\n' + someString)`, since `'\n'` is the new-line character.

A standard initializing sequence with JTextAreas is the following, unless you want to add a JScrollPane as described in Chapter Six. You do not attach a listener object to a text area because a JTextArea does not react to any events:

```
JTextArea answers = new JTextArea (50, 10);
someContainer.add (answers);
```

To re-position the scroll pane at the end of the text of `answer`, use the following:

```
answer.setCaretPosition (answer.getText().length());
```

JSliders

A **JSlider** is a bar with a **thumb** icon that moves left and right if it runs horizontally, or up and down if it runs vertically. The bar normally has a numerical scale running from 0 to 100. The `javax.swing.JSlider` class is a subclass of the JComponent class, so you can use the `setToolTipText` methods with JSliders, as well as the following:

- `new JSlider(directionInt)` creates a JSlider object that runs either horizontally or vertically, depending on the int value of the parameter. You should use either `JSlider.HORIZONTAL` or `JSlider.VERTICAL` for that parameter.
- `getValue()` returns the int value the thumb currently indicates.
- `addChangeListener(someChangeListener)` attaches a listener object to the JSlider object, which will react to any move-and-release action on the thumb.
- `setMinimum(sizeInt)` specifies the smallest value that appears on the JSlider.
- `setMaximum(sizeInt)` specifies the largest value that appears on the JSlider.
- `setMinorTickSpacing(someInt)` specifies the distance between tick marks. For instance, if the value of the parameter is 4 and the minimum is set to 10, you will have tick marks on the slider at 10, 14, 18, 22, 26, etc.

You attach a **ChangeListener** object to the JSlider to react to events with the `addChangeListener` method. When the user clicks-and-drags the thumb and then lets go, the JSlider sends a message to that ChangeListener object to execute its `stateChanged` method. ChangeListener is an interface.

This is exactly the same as what happens with a click of a JButton except that the **EventListener** used is a ChangeListener rather than an ActionListener, and the method it executes is named `stateChanged` rather than `actionPerformed`, and the parameter is a **ChangeEvent** instead of an ActionEvent. These are both subclasses of the EventObject class, which has the `getSource()` method to see which component was activated. Figure 10.7 show the EventListener classes and EventObject classes discussed so far. ChangeListener and ChangeEvent are in the `javax.swing.event` package.

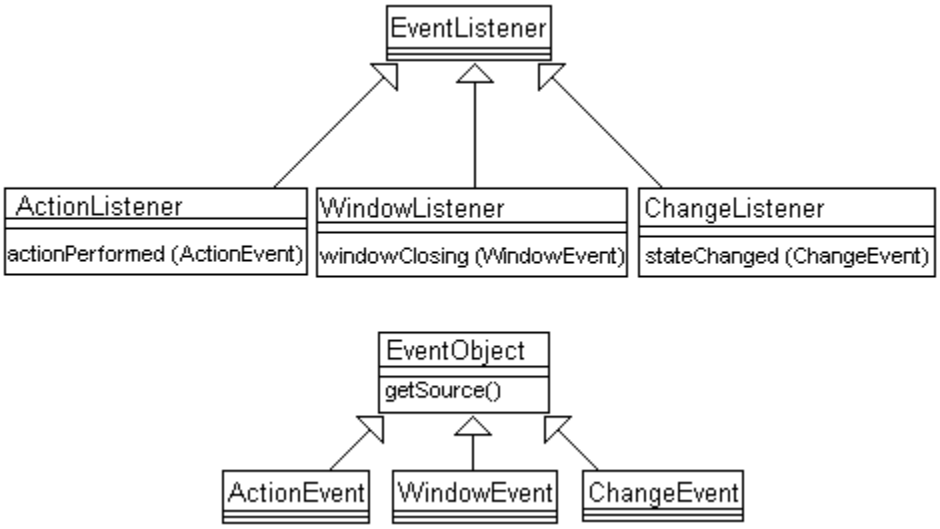

Figure 10.7 UML class diagram of listener interfaces and event classes

The constants `JSlider.HORIZONTAL` and `JSlider.VERTICAL` that indicate which way a JSlider runs are actually defined in the **SwingConstants** interface rather than the JSlider class. But the JSlider class and many others implement SwingConstants, which gives a common set of constants to all of those swing classes. An interface is allowed to specify class variables as well as instance method headings.

Listing 10.7 is an applet to illustrate the use of a JSlider and a JTextArea. Each time the user releases the thumb at some point on the slider, the corresponding value is added to a running total and displayed in a text area.

Listing 10.7 The AddInputs applet

```
import javax.swing.*;

public class AddInputs extends JApplet
{
   private JSlider itsInput = new JSlider (JSlider.HORIZONTAL);
   private JTextArea itsOutput = new JTextArea (40, 20);
   private int itsData = 0;


   public void init()
   {  setVisible (true);
      java.awt.Container content = this.getContentPane();
      content.add (itsOutput);

      itsInput.addChangeListener (new SliderCL());
      itsInput.setToolTipText ("numbers ranging 0 to 100");
      content.add (itsInput);

      JButton show = new JButton ("show total so far");
      show.addActionListener (new ButtonAL());
      content.add (show);
   }  //=====================

   private class ButtonAL
                 implements java.awt.event.ActionListener
   {
      public void actionPerformed (java.awt.event.ActionEvent ev)
      {  itsOutput.append ("\nTotal so far is " + itsData);
      }
   }  //=====================

   private class SliderCL
                 implements javax.swing.event.ChangeListener
   {
      public void stateChanged (javax.swing.event.ChangeEvent ev)
      {  itsOutput.append ("\nEntry: " + itsInput.getValue());
         itsData += itsInput.getValue();
      }
   }  //=====================
}
```

The `start` and `paint` methods are omitted because the superclass JApplet provides do-nothing implementations of them except that `paint` shows all of the components on the screen. The `init` method is executed automatically by the browser when it loads the applet. If you want to run this within a JFrame application, simply add a new AddInputs object to the JFrame's content pane and call that AddInput object's `init` method.

When you browse a web page containing this applet, it creates an AddInputs object which creates the text area, the slider, and the button and puts them on the screen. It also attaches a listener object to each of the slider and the button. Then the runtime system waits for some action:

- If the user moves the slider to the value 5 (the values range from 0 to 100), the slider sends a `stateChanged` message to whatever listener object is attached to the slider. That message tells the listener to print "Entry: 5" on the text area and add 5 to `itsData` (which is now 5).
- If the user then moves the slider to 7, the slider sends a `stateChanged` message to whatever listener object is attached to the slider, which prints a new line "Entry: 7" on the text area and adds 7 to `itsData` (which is now 12).
- If the user then clicks the button that says "show total so far", the button sends an `actionPerformed` message to whatever listener object is attached to the button. That message tells the listener to append the line "Total so far is 12" to the text area.

Figure 10.8 shows most of the relationships among the applet, content pane, button, slider, and listener objects created by Listing 10.7.

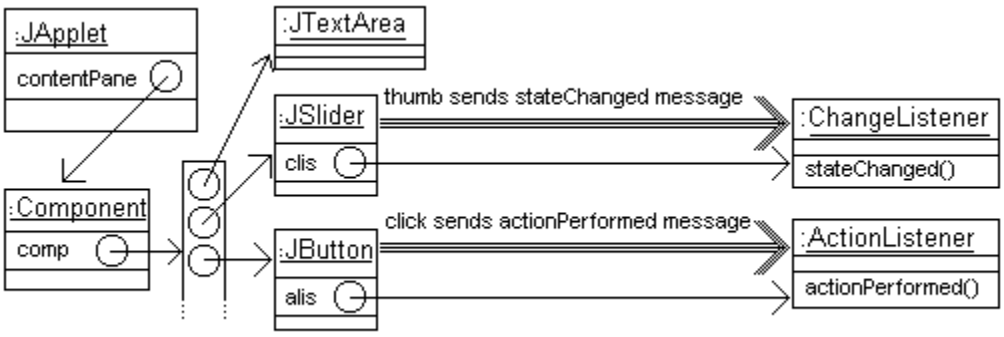

Figure 10.8 Some Component objects for Listing 10.7

A self-listener object

The kind of structure we are using for the CarRental software is not the only good one. An alternative way of organizing Components is to have a "self-listener" object for each Component-plus-listener object you want to add. That way, everything that has to do with the one component is in one location. For instance, you could replace the three statements mentioning `itsInput` in the `init` method of Listing 10.7 by the following one statement, as well as eliminate the declaration of `itsInput` itself:

```
content.add (new SliderCL());
```

All you need instead is the following larger definition of the SliderCL class, in which the JSlider is its own listener. Compare it element-by-element with Listing 10.7:

```
private class SliderCL extends JSlider
              implements javax.swing.event.ChangeListener
{  public SliderCL()
   {  super (JSlider.HORIZONTAL);
      this.addChangeListener (this);
      this.setToolTipText ("numbers ranging 0 to 100");
   }

   public void stateChanged (javax.swing.event.ChangeEvent ev)
   {  itsOutput.append ("\nEntry: " + this.getValue());
      itsData += this.getValue();
   }
}  //======================
```

Exercise 10.26 Rewrite Listing 10.7 to not have any instance variable storing the JSlider; make it a local variable instead.
Exercise 10.27 (harder) Modify Listing 10.7 to have a second slider. The values from the second one are subtracted from the running total, as if the two sliders indicated deposits and checks for a bank account.
Exercise 10.28* Rewrite Listing 10.7 to use an anonymous class in place of SliderCL.
Exercise 10.29* How would you rewrite Listing 10.7 to make ButtonAL a self-listener?
Exercise 10.30* Rewrite Listing 10.4 to use a self-listener for the JTextField.
Exercise 10.31* Read the documentation for JSliders. Then say what the initial value of the slider is and explain how to specify it at the time the slider is constructed.
Exercise 10.32* Rewrite Listing 10.4 to use a JSlider for input instead of the JTextField. Allow a maximum of 30 days total.

10.8 Swing Timers

The CarRental software clients want a timer to click off the seconds that pass while an employee is entering data. The second JPanel object still has some space on it. We will declare a subclass of JLabel named Clock and put the instance variable declaration

```
private Clock itsClock = new Clock (1000);
```

in the CarRentalView class in the earlier Listing 10.6. Then all we need do is add the following statements near the end of the `subViewTwo` method of Listing 10.6:

```
panel.add (itsClock);
itsClock.start();
```

The Timer class

The **Timer** class is in the `javax.swing` package. It has the following useful methods:

- `new Timer(waitInt, someActionListener)` creates a new Timer object that, when it is going, sends the `actionPerformed` message to the specified ActionListener parameter every `waitInt` milliseconds.
- `start()` has the Timer object start sending notifications to its ActionListener.
- `stop()` has the Timer object stop sending notifications to its ActionListener.
- `setRepeats(someBoolean)` if set to `false` has the Timer object only send one notification and then automatically stop; `true` gives multiple notifications.

For the CarRentalApp program, the Clock object should create its own private Timer object with a delay of 1000 milliseconds between notifications (i.e., one second). The text on the JLabel is the number of seconds that have passed, so the Clock object also needs an instance variable `itsCounter` to keep a record of the time that has passed.

The Clock object can be started by the statement `itsClock.start()` in the `subViewTwo` method. That starts the Timer with `itsCounter` equal to zero. Each time the Timer notifies its ActionListener object that one time period has passed, the listener increments the clock's `itsCounter` by the number of milliseconds per time period and changes its text to be that number of seconds. The Clock logic is in Listing 10.8 (see next page).

The Clock class is a separately-compiled public class rather than a private inner class because (a) it can be useful in several other programs, and (b) it has no need to access the private members of the CarRentalView class.

Listing 10.8 The Clock class

```
import javax.swing.*;

public class Clock extends JLabel
{
   private Timer itsTimer;
   private int itsMillis;  // milliseconds per activation
   private int itsCounter; // milliseconds since it was started


   public Clock (int millis)
   {  super ("0");         // create a JLabel saying "0"
      itsMillis = millis;
      itsTimer = new Timer (millis, new TimerAL());
   }  //=====================


   public void start()
   {  itsCounter = 0;
      itsTimer.start();
   }  //=====================


   public void stop()
   {  itsTimer.stop();
   }  //=====================


   private class TimerAL implements java.awt.event.ActionListener
   {
      public void actionPerformed (java.awt.event.ActionEvent ev)
      {  Clock.this.itsCounter += itsMillis;
         Clock.this.setText("" + (Clock.this.itsCounter / 1000));
      }
   }  //=====================
}
```

Restarting the clock

This design is unsatisfactory. The clock keeps ticking away, eventually recording tens of thousands of seconds as the day wears on. The clients say that the purpose of the clock is to track how long it takes the employee to make a single data entry. So the timer should restart at zero each time the employee switches to a new customer, i.e., when the clear button is clicked. And the clock should stop when the data is stored in the file, i.e., when the submit button is clicked.

Therefore, add the following statement to the `actionPerformed` method in the ClearButtonAL class of the earlier Listing 10.6:

```
itsClock.start();
```

Also add the following statement to the `actionPerformed` method in the SubmitButtonAL class in Listing 10.6:

```
itsClock.stop();
```

Timers versus busywaits

The dancing flag software in Listing 8.7 has a busywait logic, in which the CPU is kept uselessly busy doing nothing until 50 milliseconds have passed. It could be done much more nicely with a Timer. To illustrate the technique, we will apply it in a simpler case, inserting 0.1 second pauses after each horizontal line drawn by the `paint` method in Listing 10.1. We leave revising Listing 8.7 as a (hard) exercise.

Listing 10.9 breaks up the execution of the loop in Painter's `paint` method into multiple executions of the `actionPerformed` method of a Timer's listener, once each 100 milliseconds. Each tick of the Timer execute one iteration of the loop, and an instance variable `depth` keeps track of which iteration it is on (in place of the local variable `depth`). We have to be sure to stop the Timer after the last iteration. Compare the logic in Listing 10.9 closely with the original Painter class in the earlier Listing 10.1.

Listing 10.9 Time-delayed Painter class

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

public class Painter extends JFrame
{
   private Timer itsTimer = new Timer (100, new TimerAL());


   public Painter()
   {  super ("Wysiwyg Car Rentals");
      addWindowListener (new Closer());
      setSize (760, 600);   // 760 pixels wide, 600 pixels tall
      setVisible (true);
      paint (getGraphics());
   }  //=====================


   public void paint (Graphics page)
   {  page.drawString ("pattern", 10, 40);
      itsTimer.start();
   }  //======================


   private class TimerAL implements java.awt.event.ActionListener
   {
      private int depth = 40;

      public void actionPerformed (java.awt.event.ActionEvent ev)
      {  Graphics2D page = (Graphics2D) getGraphics();
         page.draw (new java.awt.geom.Line2D.Double (depth,
                    depth, depth + 30, depth));  // horizontal
         depth += 5;
         if (depth >= 580)
            itsTimer.stop();
      }
   }  //======================
}
```

Exercise 10.33 Revise Listing 10.8 so that it shows the elapsed time in tenths of a second instead of in whole seconds.

Exercise 10.34 Rewrite Listing 10.8 to not use an instance variable to keep track of the number of seconds, but still do what it originally did. Hint: Get the text that is showing and convert it to an int.
Exercise 10.35* Revise Listing 10.8 so it displays minutes and seconds, in the form illustrated by 11:46 and 4:07.
Exercise 10.36** Revise Listing 8.7 to use a Timer rather than a busywait. You will need `itsCount` as an instance variable, initialized to zero. Stop the clock when `itsCount` becomes 5. Hint: Use a boolean variable `itsDancingRight`, initialized to `true`, that keeps track of whether you are dancing to the right or to the left.

10.9 JComboBoxes Using Arrays Of Objects

The CarRental clients want to have three combo boxes where the employee chooses the month, the day, and the hour when the car will be picked up. And they want three more similar boxes for returning the car. A combo box shows all the possible choices that the employee has, in a pull-down menu with the one currently selected highlighted. If the pull-down menu is not down, the user sees only the one choice that is currently selected.

JComboBox is a subclass of JComponent. The `javax.swing.JComboBox` has the following methods (and many more besides):

- `new JComboBox(Object[])` creates a new JComboBox with the values in the array as its choices. If the array components are Strings, the words will appear as the choices, in the same order as they are stored in the array, indexed from zero up.
- `addActionListener(someActionListener)` says what listener object to notify if a selection is made.
- `getSelectedIndex()` returns the int value that is the index of the currently selected Object.
- `setSelectedIndex(idexInt)` selects the Object with the specified index.
- `getSelectedItem()` returns the Object that is currently selected.
- `setMaximumRowCount(someInt)` says how many items will be visible at a time. A scroll bar appears if this is less than the total number of items.
- `addItem(someObject)` adds this one more Object to the end of the list of choices.
- `removeItemAt(indexInt)` deletes the Object at that index from the list of choices.

The CarRental software has to have available an array of 12 Strings for the 12 months of the year. It also needs an array of Strings for the 31 days "01" through "31" and an array of Strings for the hours in the day at which people can rent a car. The client says that the permissible times are from 7 A.M. through 5 P.M., which makes 11 array entries. These class variables are declared in Listing 10.10 with array initializer lists. Figure 10.9 shows how the two new panels should look.

Listing 10.10 Arrays for ComboBoxes in the CarRentalView class

```
// CarRentalView class, part 4

   private static String[] months = {"January", "February",
         "March", "April", "May", "June", "July", "August",
         "September", "October", "November", "December"};
   private static String[] days = {"01", "02", "03", "04", "05",
         "06", "07", "08", "09", "10", "11", "12", "13", "14",
         "15", "16", "17", "18", "19", "20", "21", "22", "23",
         "24", "25", "26", "27", "28", "29", "30", "31"};
   private static String[] hours = {"7am", "8am", "9am", "10am",
         "11am", "noon", "1pm", "2pm", "3pm", "4pm", "5pm"};
```

Figure 10.9 Third and fourth panels of the CarRentalView frame

You can have instance variables named `itsStartMonth` and `itsEndMonth` to refer to the JComboBoxes that list the twelve months, declared as follows:

```
private JComboBox itsStartMonth = new JComboBox (months);
private JComboBox itsEndMonth = new JComboBox (months);
```

It is convenient for the employee if, after choosing `itsStartMonth`, the value of `itsEndMonth` automatically changes to the same month. The same should be done for the hours `itsStartHour` and `itsEndHour`. More often than not, the ending month and hour are the same as the starting month and hour, though the day is usually different, so there is no point in having the ending day change with the starting day.

This means that you need to attach an ActionListener object to `itsStartMonth` so that any change to its selected index changes the selected index of `itsEndMonth` to match. The `actionPerformed` method would therefore be coded as follows:

```
public void actionPerformed (ActionEvent ev)
{  itsEndMonth.setSelectedIndex
              (itsStartMonth.getSelectedIndex());
}
```

Similarly, you need to attach an ActionListener object to `itsStartHour` so it can update `itsEndHour` when needed. Listing 10.11 (see next page) shows the resulting additions to the CarRentalView class. Read through it to be sure you understand it all.

Drawing on a JComponent

You should not draw directly on a JApplet or JFrame if you are planning to add any components whatsoever to it. You instead draw on a JPanel that is attached to its content pane or attached to some other component. The following statement calls the standard method to refresh a drawing on a given panel object:

```
panel.paintComponent (panel.getGraphics());
```

Each JComponent has a `getGraphics` and a `paintComponent` method for this use. The `paintComponent` method is called instead of the `paint` method of a JApplet, but it does much the same thing. There is one caveat however: Be sure to have its first statement call the `paintComponent` method of the superclass. Otherwise you will not like what you see. So a sample `paintComponent` method is the following:

```
public void paintComponent (Graphics g)
{  super.paintComponent (g);
   Graphics2D page = (Graphics2D) g;
   page.draw (new Line2D.Double (20, 30, 100, 200));
   // additional drawing commands
}  //======================
```

Listing 10.11 subViewThree and subViewFour for the CarRentalView class

```
// CarRentalView class, part 5

   private JComboBox itsStartMonth = new JComboBox (months);
   private JComboBox itsStartDay   = new JComboBox (days);
   private JComboBox itsStartHour  = new JComboBox (hours);
   private JComboBox itsEndMonth   = new JComboBox (months);
   private JComboBox itsEndDay     = new JComboBox (days);
   private JComboBox itsEndHour    = new JComboBox (hours);


   public JPanel subViewThree()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 125, this.getWidth() - 20, 40);

      panel.add (new Label ("starting month/day/hour:"));
      itsStartMonth.addActionListener (new StartMonthAL());
      panel.add (itsStartMonth);
      panel.add (itsStartDay);
      itsStartHour.addActionListener (new StartHourAL());
      panel.add (itsStartHour);

      return panel;
   }  //====================


   public JPanel subViewFour()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 175, this.getWidth() - 20, 40);

      panel.add (new Label ("ending month/day/hour:"));
      panel.add (itsEndMonth);
      panel.add (itsEndDay);
      panel.add (itsEndHour);

      return panel;
   }  //====================


   private class StartMonthAL implements ActionListener
   {  public void actionPerformed (ActionEvent ev)
      {  itsEndMonth.setSelectedIndex
                   (itsStartMonth.getSelectedIndex());
      }
   }  //=====================

   private class StartHourAL implements ActionListener
   {  public void actionPerformed (ActionEvent ev)
      {  itsEndHour.setSelectedIndex
                  (itsStartHour.getSelectedIndex());
      }
   }  //=====================
```

Exercise 10.37 (harder) Write a private method that could go in the CarRentalView class and be called by `panel.add (boxOfNumbers (60))` to add a JComboBox that lists the 60 different values for the minutes part of the time ("0" through "59"). Create the array using a for-statement, not an initializer list.

10.10 JCheckBoxes, JRadioButtons, And KeyStrokes

Another piece of information that the CarRental software must record is the category of vehicle that the customer wants to rent. The four choices available are compact, full-size, luxury, and sport utility. The `subViewFive` panel is to be used for this choice. Moreover, the customer is to indicate whether he/she wants manual shift, air conditioning, and/or a CD player.

JCheckBoxes

The **JCheckBox** class is a subclass of **JToggleButton**, which describes buttons with two states, "selected" and "unselected". JToggleButton is a subclass of AbstractButton, which is also the superclass of JButton. The following methods are quite useful for `javax.swing.JCheckBox`, in addition to the `getText`, `setText`, `setMnemonic`, and `addActionListener` methods inherited from the AbstractButton class:

- `new JCheckBox(textString, someBoolean)` creates a new button with the specified text on it. The button is selected if the second parameter is `true`, unselected if `false`.
- `isSelected()` returns `true` if and only if the button is currently selected.
- `setSelected(someBoolean)` makes the button selected if the parameter is `true` and unselected otherwise.

The CarRental software is to have three JCheckBox buttons on the fifth panel, one for each of the options of "manual", "air cond", "CD player". So you could add the following three instance variables to the CarRentalView class, initialized to the most commonly chosen values:

```
JCheckBox itsManual = new JCheckBox ("manual", false);
JCheckBox itsAir = new JCheckBox ("air cond", true);
JCheckBox itsCD = new JCheckBox ("CD player", true);
```

You also need statements to attach these buttons to the fifth panel. This software does not need an `actionPerformed` method for the buttons, because no particular action is required at the time one of these buttons is selected. But you do need to keep track of the button itself so that, when the submit button is clicked to store information in the data file, its `actionPerformed` method can test `itsManual.isSelected()`, etc.

JRadioButtons

The choice among compact, full-size, luxury, and SUV could be indicated by a JComboBox, but the clients have seen JRadioButtons on web pages and they want those. Specifically, you must have a group of four buttons, one for each vehicle type, of which only one can be selected at any given time.

The **JRadioButton** class in the `javax.swing` package is a subclass of JToggleButton, so you have the `isSelected`, `setSelected`, and `addActionListener` methods available as described previously, plus the following constructor:

- `new JRadioButton(textString,someBoolean)` creates a new button with the specified text on it. The button is selected if the second parameter is `true`, unselected if `false`.

ButtonGroups

You must specify that the four particular radio buttons you have act as a group, to enforce the rule that only one can be selected at a time. Another application might have two groups of two radio buttons, which would allow one of the first two to be selected as well as one of the second two. To distinguish one way of grouping from another way, you must add the four radio buttons to a **ButtonGroup** object. The three most useful methods in the `javax.swing.ButtonGroup` class are as follows:

- `new ButtonGroup()` creates a ButtonGroup object.
- `add(someAbstractButton)` attaches the given button to the ButtonGroup executor. This method does not return a value.
- `remove(someAbstractButton)` removes the given button from the executor. This method does not return a value.

You now have enough information to add the radio buttons to the software. Declare in the CarRentalView class an additional instance variable `itsVehicle` to keep track of the kind of vehicle chosen, add to the panel a label saying to choose one of the following four buttons, and then add the four buttons to the panel. All four buttons should also be added to the same button group and all should be given the same action listener. Listing 10.12 (see next page) gives the completed coding for subViewFive, thereby completing Version 2 of the CarRental software. Figure 10.10 shows what the display looks like. Remember that the number at the far right is the clock ticking away.

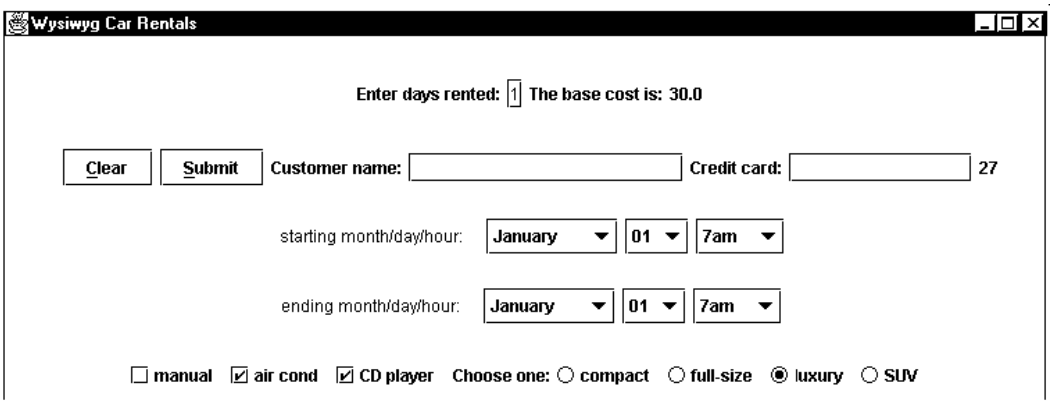

Figure 10.10 Display for the completed CarRental software

The ItemListener interface

The convention is to attach an ItemListener object to a JCheckBox instead of an ActionListener object when you want the selection to notify a listener object. The AbstractButton class has an `addItemListener` method with an ItemListener parameter for this purpose. The **ItemListener** interface requires this one method heading in any class that is to implement it:

```
public void itemStateChanged (ItemEvent ev);
```

If you wanted, for instance, the choice of a manual transmission to trigger an extra 10% discount (because people who drive a stick shift are really special people deserving of extra consideration), `itsManual.addItemListener (new TenEar())` could be used with the following inner class:

Listing 10.12 subViewFive for the CarRentalView class

```
// CarRentalView class, part 6 (completed)

   private JCheckBox itsManual = new JCheckBox ("manual", false);
   private JCheckBox itsAir = new JCheckBox ("air cond", true);
   private JCheckBox itsCD = new JCheckBox ("CD player", true);
   private String itsVehicle = "compact";  // the default choice


   public JPanel subViewFive()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 225, this.getWidth() - 20, 40);

      panel.add (itsManual);
      panel.add (itsAir);
      panel.add (itsCD);
      panel.add (new JLabel ("Choose one:"));

      ButtonGroup group = new ButtonGroup();
      ActionListener alis = new ButtonGroupAL();
      String[] text = {"compact", "full-size", "luxury", "SUV"};
      for (int k = 0;  k < text.length;  k++)
      {  JRadioButton car = new JRadioButton (text[k]);
         car.addActionListener (alis);
         group.add (car);
         panel.add (car);
      }

      return panel;
   }  //=====================


   private class ButtonGroupAL implements ActionListener
   {
      public void actionPerformed (ActionEvent ev)
      {  itsVehicle = ((JRadioButton) ev.getSource()).getText();
      }
   }  //=====================
```

```
   private class TenEar implements ItemListener
   {  public void itemStateChanged (ItemEvent ev)
      {  if (ev.getStateChange() == ItemEvent.SELECTED)
            itsDailyRate *= 0.90;
         else
            itsDailyRate /= 0.90;
      }
   }
```

You can see that an **ItemEvent** carries more information than an ActionEvent: You can query whether the event was the selecting of the button (as opposed to unselecting). If you used an ActionListener, you would have to test instead this condition:

```
((JCheckBox) ev.getSource()).isSelected().
```

Action commands

Each AbstractButton object has a private String instance variable called its "action command", which is initially the label on the button. You may set the value using `someButton.setActionCommand (someString)`. The value of this is that, when clicking a button dispatches an ActionEvent, that ActionEvent takes the action command of the button with it, which you can retrieve using `someEvent.getActionCommand()`.

Suppose that in Listing 10.12 you had four pairs of statements such as the following:

```
JRadioButton button = new JRadioButton ("compact");
button.setActionCommand ("Pontiac Sunfire");
```

Then you could determine the make of vehicle chosen within the `actionPerformed` method by referring to `ev.getActionCommand()`. For a situation where you needed to work with the make of vehicle, the alternative would be to use `((JRadioButton) ev.getSource()).getText()` in a four-part multiway selection statement, which would be clumsy and not as flexible.

Keystrokes

You may also detect keys that have been typed by the user. If you attach a KeyListener object to a textfield, for instance, each key typed within that textfield generates a KeyEvent that the KeyListener object can respond to. The preferred technique is to have an inner class that extends **KeyAdapter** (which is a Sun library implementation of the KeyListener interface that provides do-nothing methods for unneeded key operations). Then each time an ordinary key is pressed other than the shift, control, alt, or special function keys, you can get the character that was typed and take the appropriate action.

For instance, you may have the following statement in your program if you want ordinary keys that are typed within a JFrame or JApplet to have their char values (either capital or lowercase) sent to a method named `takeAppropriateAction`:

```
someJFrameOrJApplet.addKeyListener (new Respond());

private class Respond extends java.awt.event.KeyAdapter
{
   public void keyTyped (java.awt.event.KeyEvent ev)
   {  takeAppropriateAction (ev.getKeyChar());
   }
}
```

Exercise 10.38 What changes would Listing 10.12 require to have a button group of seven days of the week, recording in `itsDay` the text that is on the one chosen?
Exercise 10.39 Modify Listing 10.12 to have it select air conditioning any time the vehicle type is selected as luxury or SUV.
Exercise 10.40 Explain why you need the class cast in Listing 10.12. What class cast could you use instead of the one there?
Exercise 10.41* Revise Listing 10.12 to set a particular make and model of car as the action command for each car. Use a second array of String values so you can make the assignment within the loop.
Exercise 10.42* Revise the `actionPerformed` method of SubmitButtonAL to "add" to `itsModel` all the additional data on the third, fourth, and fifth panels.
Exercise 10.43* Read the documentation for AbstractButton and describe how to put a picture on a button.

10.11 Of Mice And Menus

If you use a modern word processor or spreadsheet, the top of the screen has a bar that says "File", "Edit", "View", etc. That is a **menu bar**. Each JFrame and JApplet has a **JMenuBar** object associated with it, at the top of the rectangular area. You can access it or replace it using the following:

```
JMenuBar bar = someJFrame.getJMenuBar();
someJFrame.setJMenuBar (someJMenuBar);
```

If you wanted a JMenuBar object put some place else on a JFrame or JApplet, you could create one using the constructor call `new JMenuBar()`. Once you have access to the JMenuBar, you can add several **JMenu** objects to it as follows:

```
bar.add (menuOne);
bar.add (menuTwo);
```

You have to first create those menus, perhaps as follows:

```
JMenu menuOne = new JMenu ("animals");
JMenu menuTwo = new JMenu ("fruits");
```

This means that the words "animals" and "fruits" will appear as the first two menus to choose from on the JMenuBar, just as the first two choices on the JMenuBar for a word-processing program are often "File" and "Edit". The `add` method in the JMenuBar class returns the JMenu object that is being added, so you could have written the preceding two pairs of statements using the `add` method, as follows:

```
JMenu menuOne = bar.add (new JMenu ("animals"));
JMenu menuTwo = bar.add (new JMenu ("fruits"));
```

Of course, a menu is no good unless it has several menu items to choose from. The following two statements put two kinds of animals on the "animals" menu:

```
menuOne.add (new JMenuItem ("dog"));
menuOne.add (new JMenuItem ("cat"));
```

Oops! There is no point in doing this, because if the user clicks on one of these kinds of animals, nothing happens. That is because you have no ActionListener to handle the event of clicking. You need to attach an ActionListener object to each item on the menu. The `add` method from the JMenu class returns the **JMenuItem** object that was added, so a compact way of adding the same ActionListener object to all MenuItems is as follows:

```
ActionListener alis = new MenuItemAL();
menuOne.add (new JMenuItem ("dog")).addActionListener (alis);
menuOne.add (new JMenuItem ("cat")).addActionListener (alis);
menuOne.add (new JMenuItem ("fish")).addActionListener (alis);
```

An alternative `add` method for JMenuItems takes a String value as the parameter, creates a JMenuItem with that text on it, and returns the JMenuItem created, so you can add two menu items to the "fruits" menu as follows:

```
menuTwo.add ("apple").addActionListener (alis);
menuTwo.add ("lemon").addActionListener (alis);
```

The JMenu class is a subclass of JMenuItem, which means that you can have a JMenu as one of the items on another menu. A menu is basically a button you click on to get a popup menu, and a menu item is basically a button you click on to get an effect (from `alis` in the example), so JMenuItem is a subclass of AbstractButton.

Reacting to a menu item choice

Now the user can select either the animals menu or the fruits menu, then click on one of the three or two choices the menu has. That sends the `actionPerformed` message to `alis`, who will need to know which MenuItem was clicked. Fortunately, `alis` can ask the ActionEvent parameter for its source, which will be the JMenuItem clicked. Then `alis` can ask that JMenuItem for its label, so it will know what action to take.

For a whimsical example, suppose the content pane contains a JLabel named `itsSound` that is to have written on it the sound that the animal makes when the user clicks an animal, or the color of the fruit when the user clicks a fruit. The `actionPerformed` method could then be as shown in Listing 10.13.

Listing 10.13 The MenuItemAL inner class of objects

```
private class MenuItemAL implements ActionListener
{
   public void actionPerformed (ActionEvent ev)
   {  String choice = ((JMenuItem) ev.getSource()).getText();
      if (choice.equals ("dog"))
         itsSound.setText ("woof");
      else if (choice.equals ("cat"))
         itsSound.setText ("meow");
      else if (choice.equals ("fish"))
         itsSound.setText ("burble");
      else if (choice.equals ("apple"))
         itsSound.setText ("red");
      else if (choice.equals ("lemon"))
         itsSound.setText ("yellow");
      else // unrecognized
         itsSound.setText ("what?");
   }
}  //======================
```

MouseListener objects (from java.awt.event)

On an unrelated note, placed here only to justify the title on this section, you can construct a listener object that reacts to user actions with the mouse. It must implement five methods of the **MouseListener** interface, to respond to each of the five possible mouse actions. The Component class (from which JComponent inherits) has a method for adding a MouseListener object:

```
someComponent.addMouseListener (someMouseListener);
```

A MouseListener object executes `mouseEntered` when the mouse cursor enters the Component and `mouseExited` when the mouse cursor leaves the Component. It executes `mousePressed` when the user presses the mouse button and `mouseReleased` when the user releases it (within the bounds of the Component). If the press and release are done without moving the mouse cursor, the MouseListener object then executes `mouseClicked` as well.

The full headings of the five methods are in Listing 10.14. Note that a special **MouseEvent** object is passed as a parameter. It allows you to retrieve the x-coordinate and y-coordinate of the point where the mouse action took place. In the example, this coordinate information is written on some label named `lab`.

Listing 10.14 The ClickML inner class of objects

```
   private class ClickML implements MouseListener
   {
      public void mouseEntered (MouseEvent ev)
      {  lab.setText ("enter " + ev.getX() + "," + ev.getY());
      }

      public void mouseExited (MouseEvent ev)
      {  lab.setText ("exit " + ev.getX() + "," + ev.getY());
      }

      public void mousePressed (MouseEvent ev)
      {  lab.setText ("press " + ev.getX() + "," + ev.getY());
      }

      public void mouseReleased (MouseEvent ev)
      {  lab.setText ("release " + ev.getX() + "," + ev.getY());
      }

      public void mouseClicked (MouseEvent ev)
      {  lab.setText ("click " + ev.getX() + "," + ev.getY());
      }
   }  //======================
```

Adapters

You may replace `implements MouseListener` by `extends MouseAdapter` in the ClickML class heading. **java.awt.event.MouseAdapter** is a convenience class in Java that implements all five methods with do-nothing coding, so a subclass can just override the ones it wants. This is nice when you only want to react to one or two of the five possible events. There is also a **java.awt.event.WindowAdapter** class that has do-nothing coding for all seven WindowListener methods (Listing 10.2) for the same purpose, as well as the java.awt.event.KeyAdapter class mentioned in the previous section.

Exercise 10.44 Add a third menu named "cars" to the `bar` JMenuBar. Then add three brands of cars to that menu, each with the same ActionListener as all other items.
Exercise 10.45 Add logic to Listing 10.13 to have a click on one of the three kinds of cars added by the preceding exercise display on `itsSound` an alliterative adjective describing the car.
Exercise 10.46 Rewrite Listing 10.2 to use a WindowAdapter instead.
Exercise 10.47* Read the documentation for JMenuBar and describe how you would find out the current number of menus on the `bar` JMenuBar.
Exercise 10.48** Read the documentation for JMenu and describe how you would remove the "cat" from the "animals" menu and leave the other two.
Exercise 10.49** Revise Listing 10.13 to use `getActionCommand` appropriately. Also make the other changes needed to store the action commands in the five menu items.

*10.12 More On LayoutManagers And JLists (*Sun Library)*

The only way you have seen to control the layout of Components is FlowLayout (from the `java.awt` package), which puts things in book-reading order one row at a time, and `setBounds` to specify precisely the location and size of a Component when the LayoutManager is null. It is easier to use one of the several LayoutManager objects available when it gives you what you want. The following methods are useful with any of the LayoutManagers described here:

- `someContainer.setLayout(someLayoutManager)` specifies the LayoutManager to be used by that Container.
- `someContainer.getLayout()` returns the LayoutManager currently in use.
- `someLayoutManager.layoutContainer(someContainer)` has the layout's container recalculate the layouts when an addition or deletion is made. This method is in the **LayoutManager** interface that all layout managers implement.
- `Box.createGlue()` returns a Component that expands to fill any available excess space. This lets you avoid having all excess space at the end of a layout; it is a class method in the `javax.swing.Box` class.

BorderLayout objects (from java.awt)

The content pane of a JFrame or JApplet is initially given a BorderLayout. You may add up to five Components to a Container that has a BorderLayout, one in each of the NORTH, SOUTH, EAST, WEST, or CENTER of the area. A Component in the NORTH or SOUTH part extends horizontally for the entire width of the area. A Component in the EAST or WEST usually takes up about a third of the width. A Component in the CENTER expands to take up whatever area is left.

- `new BorderLayout(x, y)` constructs a layout manager with `x` pixels between Components horizontally and `y` pixels between them vertically. Both parameters are int values.
- `someContainer.add(someComponent, BorderLayout.NORTH)` puts the Component in the NORTH position (and similarly for the other four positions) when using BorderLayout. Leaving out the second parameter puts the Component in the CENTER. If you try to put two Components in one part, it could foul things up.

BoxLayout objects (from javax.swing)

A BoxLayout puts all its components in one row, either horizontally or vertically as you choose. It makes them all the same height (respectively, width) to the extent possible. Different Containers cannot share the same BoxLayout object.

- `new BoxLayout(someContainer, BoxLayout.X_AXIS)` constructs a manager with a horizontal row of Components.
- `new BoxLayout(someContainer, BoxLayout.Y_AXIS)` constructs a manager with a vertical column of Components.

GridLayout objects (from java.awt)

A GridLayout object divides the area into a rectangular grid of subareas, all of the same size. Components are added left-to-right in the first row, then in the next row, etc. You may specify the number of rows to be zero, which means that the layout will allow as many rows as are needed for the Components you add (and similarly for columns, but you cannot have both values zero). Note that specifying 1 for the rows or columns gives roughly the equivalent of a BoxLayout.

- `new GridLayout(numRows, numCols)` constructs a manager with `numRows` rows and `numCols` columns.
- `new GridLayout(numRows, numCols, x, y)` constructs a manager with `numRows` rows and `numCols` columns, and with `x` pixels between Components horizontally and `y` pixels between Components vertically. All parameters are ints.

CardLayout objects (from java.awt)

The Components added in a CardLayout do not all appear in its Container; only one appears at a time, initially the first one added to the Container. The CardLayout class has several methods for switching in another Component to display. The ordering of the Components is the order in which the `add` method was called for them.

- `new CardLayout()` constructs a manager with only one Component showing.
- `someCardLayout.first(someContainer)` displays the first Component.
- `someCardLayout.last(someContainer)` displays the last Component.
- `someCardLayout.next(someContainer)` displays the next Component after the one currently displayed.
- `someCardLayout.previous(someContainer)` displays the Component before the one currently displayed.

JList objects (from javax.swing)

JList is a subclass of JComponent that lists a number of Objects. You specify the Objects as an `Object[]` parameter to the JList constructor. The order of the Objects displayed is the order they occur in the array (e.g., component 3 of the array is at index 3 in the JList and will be the fourth Object displayed).

- `new JList(someArray)` constructs a JList with as many items on it as the array has. `someArray` must be an array of Objects (or a subclass of Object).
- `someJList.setVisibleRowCount(someInt)` specifies how many values will appear at a time. However, you do not get a scroll bar with it, so you generally want to also say `someContainer.add(new JScrollPane(someJList))`.
- `someJList.getSelectedIndex()` returns the index number of the Object currently selected. It returns -1 if no value is currently selected.
- `someJList.getSelectedValue()` returns the Object currently selected, or null if nonesuch.
- `someJList.setSelectedIndex(someInt)` makes that Object the one currently selected.
- `someJList.addListSelectionListener(someListSelectionListener)` adds a listener object that reacts whenever a selection is made.

You may have a JList that allows selection of several items from the list at one time. But the default (when the JList object is constructed) is to allow just one selection. Read your Java documentation if you want to find out how to handle multiple selections.

A typical kind of listener class for a JList is the following:

```
private class MyListEar implements ListSelectionListener
{  public void valueChanged (ListSelectionEvent ev)
   {  // whatever action is appropriate
   }
}  //======================
```

*10.13 More On JComponent, ImageIcon, And AudioClip (*Sun Library)*

You may set the Font object that controls how letters and other characters look for any JComponent object such as a JTextField or JLabel (Fonts were discussed at the end of Chapter Eight). You only need the following kind of statement:

```
someJComponent.setFont (someFont);
```

You may want to put a border on a JComponent, particularly if it is a JPanel. The simplest way to do this is with the following statement, which places a simple green line around a component. Look in the Java documentation for more kinds of borders:

```
someJComponent.setBorder
       (BorderFactory.createLineBorder (Color.green));
```

ImageIcon objects (from javax.swing)

Suppose you have a small picture in a disk file named "oakTree.gif" in the plants folder. You can get it into your program as follows:

```
ImageIcon tree = new ImageIcon ("plants/oakTree.gif");
```

Now you can display the picture in any of several ways. One way is to just put it at the `<x,y>` coordinates where you want it on a particular component using its Graphics:

```
tree.paintIcon (someComponent, someGraphics, x, y);
```

Another way is to put it on a JLabel or other JComponent, along with whatever other text might be there (see your JLabel documentation for how to control which goes where):

```
someJComponent.setIcon (tree);
```

You can also create an Image object on which you can draw and then display on a component:

- `Image p = someComponent.createImage (width, height)` returns a new Image on which you can draw.
- `p.getGraphics()` returns the Graphics context for that Image.
- `someGraphics.draw (p, x, y, someComponent)` draws the Image on `someComponent` at position (x,y) using that Component's Graphics context.

AudioClip objects (interface from java.applet)

An Applet (and thus a JApplet) has two methods that help you play sounds: `getDocumentBase` returns the URL of the place where the HTML file was loaded from, and `getAudioClip` returns an AudioClip object found at a particular URL. So you only need a statement of the following form to get some music into your Applet:

```
AudioClip music = getAudioClip (getDocumentBase(), "pop.au");
```

The AudioClip interface prescribes three methods to let you control the sound of `music`:

- `music.play()` plays the sound one time through.
- `music.loop()` plays the sound over and over again.
- `music.stop()` stops playing the sound.

10.14 Review Of Chapter Ten

Listing 10.5 illustrates inner classes, the only new language feature introduced in this chapter other than anonymous classes, which should be used quite sparingly anyway.

About the Java language:

- If class X **implements** Y, then X must override every method prescribed by Y. Your coding is not allowed to attempt to execute the body of a method in Y itself.
- You can declare a class `private class X` inside of an outer class Out. You then construct an object of this **inner class** X only with an Out object as the executor, as in `someOut.new X()`. Of course, if the construction takes place inside an instance method or constructor of the Out class, the Out object defaults to the executor.
- Inside class X, you may refer to the Out object that constructed the X executor of a method as `Out.this`. A reference to an instance variable or method inside X that only makes sense with an Out object as executor defaults to `Out.this`. An inner class cannot have any class variable or class method.
- To make an **anonymous class**, put the body of the class directly after the parentheses of a constructor call `new Whatever()`, with no heading on the class. The class is considered to be a subclass of Whatever, or an implementation if Whatever is an interface. The anonymous class is considered an inner class; it cannot contain constructors or class methods or class variables. It is bad form to have more than half-a-dozen statements total inside an anonymous class.
- **Event-driven programming** is attaching listener objects to components (**registering** the event-handler) and thereafter only taking action in response to an event that causes one of those components to send an action message to its listener object.
- Read through the documentation for the `java.awt` and `javax.swing` classes partially described in this chapter. Look at the API documentation at `http://java.sun.com/docs` or on your hard disk.

About the java.awt.Component class (has no constructors):

- `someComponent.paint(someGraphics)` paints this Component.
- `someComponent.repaint()` colors over everything in the background color and then calls paint.
- `someComponent.getGraphics()` returns the Component's Graphics object.
- `someComponent.setSize(widthInt, heightInt)` resizes it to the given width and height.
- `someComponent.setVisible(someBoolean)` makes it visible or invisible with its existing Components. Components added to it later may not appear, so `setVisible(true)` should usually be the last thing you do when constructing a JFrame, except perhaps call `paint`.
- `someComponent.setBounds(xInt, yInt, widthInt, heightInt)` puts its upper-left corner at `<xInt,yInt>` and makes its size `<widthInt,heightInt>`. This only takes effect if the LayoutManager of its Container is set to null.
- `someComponent.getWidth()` returns the int width of the Component in pixels.
- `someComponent.getHeight()` returns the int height of the Component in pixels.

About the java.awt.Container class (a subclass of Component; includes six useful methods not mentioned earlier in the text):

- `someContainer.add(someComponent)` adds the parameter to the end of a list of Components attached to this Container and returns the Component that was added.
- `someContainer.setLayout(someLayoutManager)` specifies the way the Components will be laid out in this Container. The LayoutManager must be set to null if its Components are to use `setBounds`.

- `someContainer.getComponentCount()` returns the number of components attached to this container.
- `someContainer.getComponent(index)` returns the component indexed as specified (using zero-based indexing, as usual).
- `someContainer.add(someComponent, index)` inserts `someComponent` at that index.
- `someContainer.remove(index)` removes the component with that index.
- `someContainer.removeAll()` removes all of its components.
- `someContainer.validate()` lays out the components anew. This is useful after adding or removing any components.

About the javax.swing.JFrame class (a subclass of Container):

- `new JFrame(titleString)` creates an initially invisible window on the monitor screen with the specified title.
- `someFrame.getTitle()` returns its title (a String at the top of the frame).
- `someFrame.setTitle(titleString)` replaces a JFrame object's title.
- `someFrame.addWindowListener(someWindowListener)` attaches a listener.
- `someFrame.getContentPane()` returns the Container object to which all components of a JFrame are to be attached. Also available for a JApplet.
- `someFrame.getJMenuBar()` returns its menu bar. Also available for a JApplet.
- `someFrame.setJMenuBar(JMenuBar)` replaces its menu bar. Also available for a JApplet.

About the javax.swing.JComponent class (a subclass of Container):

- JComponent is designed for a pluggable look-and-feel.
- `someJComponent.setToolTipText(flyoverString)` specifies the text that pops out when the mouse cursor passes over it.

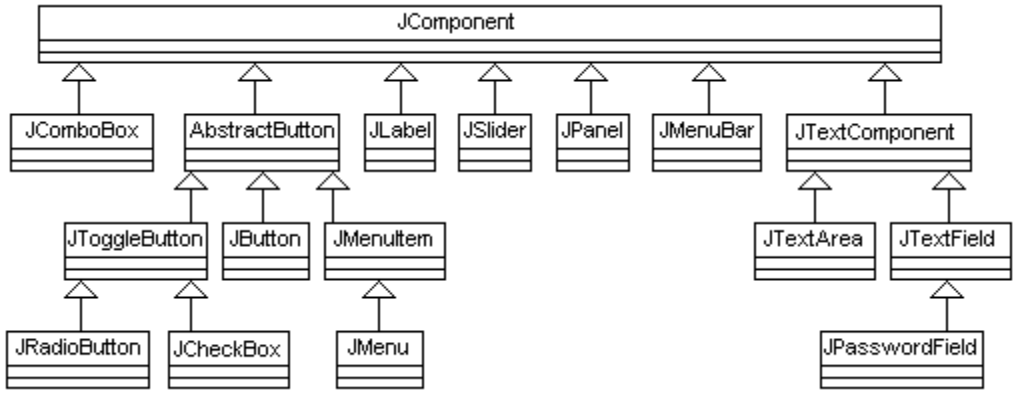

Figure 10.11 Subclasses of JComponent discussed in this chapter

Constructors for various JComponent subclasses (all are in javax.swing):

- `new JPanel()`. JPanel is a subclass of JComponent.
- `new JLabel(textString)`. JLabel is a subclass of JComponent.
- `new JTextField(textString)`. JTextField a subclass of JTextComponent. A JTextField allows just one line of input text and reacts to the ENTER key.
- `new JTextField(widthInt)` specifies the width in 'm' characters.
- `new JPasswordField(widthInt)` specifies the width in 'm' characters.
- `new JButton(textString)`. JButton is a subclass of AbstractButton.

- `new JTextArea(rowInt, columnInt)`. JTextArea is a subclass of JTextComponent. It is described in some detail in Chapter Six.
- `new JSlider(directionInt)`. JSlider is a subclass of JComponent. The int parameter can be `JSlider.HORIZONTAL` or `JSlider.VERTICAL`.
- `new JComboBox(Object[])`. JComboBox is a subclass of JComponent. The given list of Objects tells the choices available, in the order listed.
- `new JCheckBox(textString, someBoolean)`. JCheckBox is a subclass of AbstractButton. The optional second parameter makes its selected status `true` or `false`.
- `new JRadioButton(textString, someBoolean)`. JRadioButton is a subclass of AbstractButton. The optional second parameter makes its selected status `true` or `false`. In a ButtonGroup of buttons, only one button can be selected.
- `new JMenuBar()`. JMenuBar is a subclass of JComponent.
- `new JMenu(textString)`. JMenu is a subclass of JMenuItem.
- `new JMenuItem(textString)`. JMenuItem is a subclass of AbstractButton.

Methods that add listener objects to subclasses of JComponents:

- `comp.addActionListener(someActionListener)` is in the AbstractButton, JTextField, and JComboBox classes, and thus is also available for JButton, JRadioButton, JCheckBox, and JMenuItem objects.
- `someJSlider.addChangeListener(someChangeListener)` is in the JSlider class.
- `someAbstractButton.addItemListener(someItemListener)` is in the AbstractButton class, but we generally only use it for a JCheckBox object.
- `someComponent.addKeyListener(someKeyListener)` is in the Component class and therefore available for all kinds of JComponents, especially JFrames.
- `someComponent.addMouseListener(someMouseListener)` is in the Component class and therefore available for all kinds of JComponents.

Other quite useful methods for subclasses of JComponent:

- `comp.setText(textString)` changes the text on the component. It is in JLabel, AbstractButton, and JTextComponent, and thus available for JButton, JRadioButton, JCheckBox, JMenuItem, JTextField, and JTextArea objects.
- `comp.getText()` returns the current text on the component. It has the same availability as the preceding `setText`.
- `someJPasswordField.getPassword()` returns a char array.
- `someAbstractButton.setMnemonic(someChar)` makes that character the hotkey for that AbstractButton, so that the ALT key followed by this character effectively clicks or selects that button.
- `someJTextArea.append(someString)` adds text to the end of the existing text in the JTextArea.
- `someJSlider.getValue()` returns the int value where the user released the thumb of a JSlider.
- `someJSlider.setMinimum(sizeInt)` specifies the smallest value that appears on a JSlider.
- `someJSlider.setMaximum(sizeInt)` specifies the largest value that appears on a JSlider.
- `someJSlider.setMinorTickSpacing(someInt)` specifies the distance between ticks on a JSlider.
- `someJComboBox.getSelectedIndex()` returns the int value that is the index of the currently selected Object in a JComboBox.
- `someJComboBox.setSelectedIndex(indexInt)` selects the Object with the specified index in a JComboBox.
- `someJComboBox.getSelectedItem()` returns the Object currently selected in a JComboBox.

- `someJComboBox.setMaximumRowCount(someInt)` gives the number of items displayed by a JComboBox, with a scroll bar if the actual number of items is greater.
- `someJComboBox.addItem(someObject)` adds this Object to the end of the list for a JComboBox.
- `someJComboBox.removeItemAt(indexInt)` deletes the Object at that index from the JComboBox.
- `comp.isSelected()` tells whether the JRadioButton or JCheckBox is currently selected.
- `comp.setSelected(someBoolean)` makes the JRadioButton or JCheckBox selected if the parameter is `true` and makes it unselected otherwise.
- `comp.setActionCommand(actionString)` stores a String value in the JComboBox, JTextfield, or AbstractButton (so it can be retrieved from an ActionEvent object for JButton, JRadioButton, JCheckBox, JMenu, or JMenuItem objects).
- `someJMenuBar.add(someJMenu)` adds the next menu to the MenuBar and returns that JMenu object.
- `someJMenu.add(someJMenuItem)` adds the next menu item to the JMenu and returns the JMenuItem that was added.
- `someJMenu.add(someString)` adds to the JMenu a new JMenuItem that it constructs from the String, and returns the JMenuItem that was added.

About the javax.swing.Timer class (a direct subclass of Object):

- `new Timer(waitInt, someActionListener)` creates a Timer object that, when running, sends an `actionPerformed` message to the ActionListener parameter every `waitInt` milliseconds.
- `someTimer.start()` has the Timer object start sending notifications to its listener.
- `someTimer.stop()` has the Timer object stop sending notifications to its listener.
- `someTimer.setRepeats(someBoolean)` if set to `false` has the Timer object only send one notification and then stop; `true` gives multiple notifications.

About the javax.swing.ButtonGroup class (a direct subclass of Object):

- `new ButtonGroup()` creates a ButtonGroup object to which JRadioButtons can be attached. When a button in that group is selected, the ButtonGroup object will deselect all other buttons in that group. A ButtonGroup is not a Component.
- `someButtonGroup.add(someAbstractButton)` attaches the given button to the ButtonGroup executor. This method does not return a value.
- `someButtonGroup.remove(someAbstractButton)` removes the given button from the ButtonGroup executor. This method does not return a value.

About EventObject and its subclasses:

- `EventObject` (in `java.util`) has the method `getSource()` that returns the Object that generated the event. It has the following five among its subclasses:
- `WindowEvent` (in `java.awt.event`) is used for JFrames.
- `ActionEvent` (in `java.awt.event`) has the method `getActionCommand()` for retrieving the action command String value attached to the object that generated the event (if any).
- `ChangeEvent` (in `javax.swing.event`) is used for JSliders.
- `ItemEvent` (in `java.awt.event`) has the `getStateChange()` method that either returns `ItemEvent.SELECTED` (to indicate that the object was selected) or `ItemEvent.DESELECTED` (so it was not selected).
- `KeyEvent` (in `java.awt.event`) has the `getKeyChar()` method for finding the char value of the key that was pressed, other than special keys such as alt and shift.
- `MouseEvent` (in `java.awt.event`) has the `getX()` and `getY()` methods for finding the `<x,y>` coordinates where the mouse action occurred.

About listener interfaces (all are sub-interfaces of java.util.EventListener):

- The `WindowListener` interface (in `java.awt.event`) has seven methods, one of which is `windowClosing(someWindowEvent)`, called when the user clicks on the window's closer icon.
- The `ActionListener` interface (in `java.awt.event`) has one method `actionPerformed(someActionEvent)` that is called when a Component experiences an event that its ActionListener object can respond to.
- The `ChangeListener` interface (in `javax.swing.event`) has one method `stateChanged(someChangeEvent)` that is called when a Component experiences an event that its ChangeListener object can respond to.
- The `ItemListener` interface (in `java.awt.event`) has one method `itemStateChanged(someItemEvent)` that is called when an item changes its state.
- The `KeyListener` interface (in `java.awt.event`) has three methods, one of which is `keyPressed(someKeyEvent)` that is called when an ordinary key is pressed. Extend `java.awt.event.KeyAdapter` to avoid the other two methods.
- The `MouseListener` interface (in `java.awt.event`) has five methods that are called when various mouse actions occur. The one that reacts to the click of the mouse has the heading `public void mouseClicked (MouseEvent ev)`. The other four have the same heading except "Clicked" is replaced by "Entered", "Exited", "Pressed", and "Released", respectively. The MouseEvent object has two methods `ev.getX()` and `ev.getY()` to find where the action took place.

Answers to Selected Exercises

10.1 Delete "javax.swing." and "java.awt." and "java.awt.geom." in one place each in the coding.
Instead, put the following three phrases at the top of the file:
import javax.swing.JFrame; import java.awt.Graphics; import java.awt.geom.Line2D;

10.2 The lower-right corner of the drawing, for depth=575, is <605, 575>, so setSize (616, 585).

10.7 The text field that initially contains the number 1 would appear to the left of the phrase "Enter days rented:" on the display. But putting the adding of the textfield to the panel before the adding of the ActionListener makes no difference.

10.8 Add the following statement just before the return statement:
panel.add (new JLabel ("for a compact"));

10.9
```
public void actionPerformed (ActionEvent ev)
{   double d = Double.parseDouble (itsDaysRented.getText());
    itsBaseCost.setText ("" + (d - 32.0) * 5 / 9);
}
```

10.11 Replace the following: "private JPanel subViewOne() { JPanel panel = new JPanel();"
by this: "private class SubViewOne extends JPanel { public SubViewOne() {"
Replace "this" by "CarRentalView.this" in two places.
Replace "panel" by "this" throughout the method, except replace "return panel;" by "}".

10.12 A bad answer for this problem is to make itsDaysRented and itsBaseCost public variables.
Instead, add to CarRentalView two methods getDaysRentedText() and setBaseCostText(String) with the obvious one statement of coding each and have the following statement in subViewOne:
itsDaysRented.addActionListener (new DaysRentedAL (this));
Then code the DaysRentedAL class as follows:

```
public class DaysRentedAL implements ActionListener
{   private CarRentalView itsView;
    public DaysRentedAL (CarRentalView givenView)
    {   itsView = givenView;
    }
    public void actionPerformed (ActionEvent ev)
    {   int d = Integer.parseInt (itsView.getDaysRentedText());
        itsView.setBaseCostText ("" + (d - d / 7) * CarRentalView.PRICE_PER_DAY);
    }
}
```

10.15 Add the following statement to the actionPerformed method:
((JButton) ev.getSource()).setText ("Ouch!");

10.16 Replace the two lines beginning with "if" by the following:

```
if (source.getText().equals ("X"))
      JOptionPane.showMessageDialog (null, "You chose that one before");
else if (source.getText().equals ("O"))
      JOptionPane.showMessageDialog (null, "That one belongs to me already");
```

10.18 It would appear centered below the row that contains the other components.

10.19 Move the two statements that add the buttons to be at the end of the subViewTwo method, just before the return statement.

10.20

```
public void actionPerformed (ActionEvent ev)
{     if (itsCustomerName.getText().length() == 0 || itsCreditCard.getText().length() == 0)
            System.out.println ("You cannot have a blank entry");
      else
            itsModel.add (itsCustomerName.getText() + "  " + itsCreditCard.getText());
}
```

10.26 Move the declaration of itsInput to right after content.add (itsOutput), omitting "private":

```
JSlider itsInput = new JSlider (JSlider.HORIZONTAL);
```

In the stateChanged method, insert this as the first statement:

```
JSlider itsInput = (JSlider) ev.getSource();
```

10.27 Define another instance variable:

```
private JSlider itsNeg = new JSlider (JSlider.HORIZONTAL);
```

Put the following two statements in the AddInputs init method:

```
content.add (itsNeg);
itsNeg.addChangeListener (new SliderCL());
```

Replace the stateChanged method by the following:

```
public void stateChanged (ChangeEvent ev)
{     int amount = event.getSource() == itsInput  ?  itsInput.getValue()  :  - itsMinus.getValue();
      itsOutput.append ("\nEntry: " + amount);
      itsData += amount;
}
```

10.33 Change the String parameter in the last statement of the actionPerformed method to the following:

```
(Clock.this.itsCounter / 1000) + "." + (Clock.this.itsCounter / 100 % 10).
```

10.34 Replace the first statement of the start method by setText ("0");

Replace the two statements of the actionPerformed method by the following:

```
Clock.this.setText ("" + (1 + Integer.parseInt (Clock.this.getText())));
```

10.37

```
private JComboBox boxOfNumbers (int howMany)
{     String[ ] values = new String [howMany];
      for (int k = 0;  k < howMany;  k++)
            values[k] = "" + k;
      return new JComboBox (values);
}
```

10.38 You only need to rename itsVehicle as itsDay and write the declaration of text as follows:

```
String[ ] text = {"Sun", "Mon", "Tue", "Wed", "Thur", "Fri", "Sat"};
```

10.39 Add this statement as the last statement of ButtonGroupAL's actionPerformed method:

```
if (itsVehicle.equals ("luxury") || itsVehicle.equals ("SUV"))
      itsAir.setSelected (true);
```

10.40 ev.getSource() returns an Object value, not a JRadioButton value.

So ev.getSource() does not have a getText method. So a cast is needed.

Since the getText method is actually in the AbstractButton class, you could cast it to AbstractButton or to JToggleButton and get the same effect.

10.44

```
JMenu menuTres = bar.add (new JMenu ("cars"));
menuTres.add ("Ford").addActionListener (alis);
menuTres.add ("Chevrolet").addActionListener (alis);
menuTres.add ("Lexus").addActionListener (alis);
```

10.45 Insert the following just before the last "else" of the method in Listing 10.13:

```
else if (choice.equals ("Ford"))
      itsSound.setText ("fantastic Ford");
else if (choice.equals ("Chevrolet"))
      itsSound.setText ("shiny Chevrolet");
else if (choice.equals ("Lexus"))
      itsSound.setText ("lovely Lexus");
```

10.46

```
public class Closer extends WindowAdapter
      {     public void windowClosing (WindowEvent ev)
            {     System.exit (0);
            }
      }
```