CHAPTER 14

Introduction to Swing Components

In this chapter, you will:

- Understand Swing components
- Use the JFrame class
- Use the JLabel class
- Use a layout manager
- Extend the JFrame class
- Add JTextFields, JButtons, and tool tips to a JFrame
- Learn about event-driven programming
- Understand Swing event listeners
- Use the JCheckBox, ButtonGroup, and JComboBox classes

Understanding Swing Components

Computer programs usually are more user friendly (and more fun to use) when they contain user interface (UI) components. **UI components** are buttons, text fields, and other components with which the user can interact. Java's creators have packaged a number of prewritten components in the Swing package. **Swing components** are UI elements such as dialog boxes and buttons; you can usually recognize their names because they begin with *J*.

Swing components were named after a musical style that was popular in the 1940s. The name is meant to imply that the components have style and pizzazz. You have already used the JOptionPane component that is part of the Swing class. The Swing classes are part of a more general set of UI programming capabilities that are collectively called the **Java Foundation Classes**, or **JFC**. JFC includes Swing component classes and selected classes from the java.awt package.

In early versions of Java, components had simple names, such as Frame and Button. The components created from these original classes did not have a consistent appearance when used with different browsers and operating systems. When Java's creators designed new, improved classes, they needed new names for the classes, so they used a *J* in front of each new class name. Hence, Swing components have names like JFrame, JButton, JScrollbar, JOptionPane, and so on.

UI components are also called *controls* or *widgets*. Each Swing component is a descendant of a JComponent, which in turn inherits from the java.awt.Container class. You can insert the statement `import javax.swing.*;` at the beginning of your Java program files so you can take advantage of the Swing UI components and their methods. When you import Swing classes, you use the javax.swing package instead of java.swing. The *x* originally stood for *extension*, so named because the Swing classes were an extension of the original Java language specifications.

Almost all Swing components are said to be **lightweight components** because they are written completely in Java and do not have to rely on the local operating system code. This means the components are not "weighed down" by having to interact with the operating system (for example, Windows or Macintosh) in which the application is running. Some Swing components, such as JFrames, are known as **heavyweight components** because they do require interaction with the local operating system. A lightweight component reuses the native (original) window of its closest heavyweight ancestor; a heavyweight component has its own opaque native window. The only heavyweight components used in Swing are swing.JFrame, swing.JDialog, swing.JWindow, swing.JApplet, awt.Component, awt.Container, and awt.JComponent.

When you use Swing components, you usually place them in containers. A **container** is a type of component that holds other components so you can treat a group of them as a single entity. Containers are defined in the Container class. Often, a container takes the form of a window that you can drag, resize, minimize, restore, and close.

As you know from reading about inheritance in Chapters 10 and 11, all Java classes descend from the Object class. The Component class is a child of the Object class, and the Container class is a child of the Component class. Therefore, every Container object "is a" Component, and every Component object (including every Container) "is an" Object. The Container class is also a parent class, and the Window class is a child of Container. However, Java programmers rarely use Window objects because the Window subclass Frame and its child,

the Swing component **JFrame**, both allow you to create more useful objects. Window objects do not have title bars or borders, but JFrame objects do.

TWO TRUTHS & A LIE

Understanding Swing Components

1. Swing components are elements such as buttons; you can usually recognize their names because they contain the word *Swing*.
2. Each Swing component is a descendant of a JComponent, which in turn inherits from the java.awt.Container class.
3. You insert the import statement `import javax.swing.*;` at the beginning of your Java program files so you can use Swing components.

The false statement is #1. You can usually recognize Swing component names because they begin with *J*.

Using the JFrame Class

You usually create a JFrame so that you can place other objects within it for display. Figure 14-1 shows the JFrame's inheritance tree. Recall that the Object class is defined in the java.lang package, which is imported automatically every time you write a Java program. However, Object's descendants (shown in Figure 14-1) are not automatically imported.

```
java.lang.Object
   |--java.awt.Component
         |--java.awt.Container
               |--java.awt.Window
                     |--java.awt.Frame
                           |--javax.swing.JFrame
```

Figure 14-1 Relationship of the JFrame class to its ancestors

The JFrame class has four constructors:

- `JFrame()` constructs a new frame that initially is invisible and has no title.
- `JFrame(String title)` creates a new, initially invisible JFrame with the specified title.
- `JFrame(GraphicsConfiguration gc)` creates a JFrame in the specified GraphicsConfiguration of a screen device with a blank title. (You will learn about the GraphicsConfiguration class as you continue to study Java.)

- `JFrame(String title, GraphicsConfiguration gc)` creates a `JFrame` with the specified title and the specified `GraphicsConfiguration` of a screen.

You can construct a `JFrame` as you do other objects, using the class name, an identifier, the assignment operator, the `new` operator, and a constructor call. For example, the following two statements construct two `JFrames`: one with the title "Hello" and another with no title:


```
JFrame firstFrame = new JFrame("Hello");
JFrame secondFrame = new JFrame();
```

After you create a `JFrame` object, you can use the now-familiar object-dot-method format you have used with other objects to call methods that manipulate a `JFrame`'s features. Table 14-1 describes some useful `JFrame` class methods.

The methods in Table 14-1 represent only a small portion of the available methods you can use with a `JFrame`. Each of the methods listed in Table 14-1 is inherited from either `JFrame`'s `Component` or `Frame` parent class. These classes contain many useful methods in addition to the few listed here. You can read the documentation for all the methods at *www.oracle.com/technetwork/java/index.html*.

Method	Purpose
`void setTitle(String)`	Sets a `JFrame`'s title using the `String` argument
`void setSize(int, int)`	Sets a `JFrame`'s size in pixels with the width and height as arguments
`void setSize(Dimension)`	Sets a `JFrame`'s size using a `Dimension` class object; the `Dimension(int, int)` constructor creates an object that represents both a width and a height
`String getTitle()`	Returns a `JFrame`'s title
`void setResizable(boolean)`	Sets the `JFrame` to be resizable by passing `true` to the method, or sets the `JFrame` not to be resizable by passing `false` to the method
`boolean isResizable()`	Returns `true` or `false` to indicate whether the `JFrame` is resizable
`void setVisible(boolean)`	Sets a `JFrame` to be visible using the `boolean` argument `true` and invisible using the `boolean` argument `false`
`void setBounds(int, int, int, int)`	Overrides the default behavior for the `JFrame` to be positioned in the upper-left corner of the computer screen's desktop. The first two arguments are the horizontal and vertical positions of the `JFrame`'s upper-left corner on the desktop. The final two arguments set the width and height.

Table 14-1 Useful methods inherited by the `JFrame` class

Assuming you have declared a JFrame named firstFrame, you can use the following statements to set the firstFrame object's size to 250 pixels horizontally by 100 pixels vertically and set the JFrame's title to display a String argument. **Pixels** are the picture elements, or tiny dots of light, that make up the image on your computer monitor.

```
firstFrame.setSize(250, 100);
firstFrame.setTitle("My frame");
```

When you set a JFrame's size, you do not have the full area available to use because part of the area is consumed by the JFrame's title bar and borders.

Figure 14-2 shows an application that creates a small, empty JFrame.

```
import javax.swing.*;
public class JFrame1
{
   public static void main(String[] args)
   {
      JFrame aFrame = new JFrame("First frame");
      aFrame.setSize(250, 100);
      aFrame.setVisible(true);
   }
}
```

Figure 14-2 The JFrame1 application

The application in Figure 14-2 produces the JFrame shown in Figure 14-3. It resembles frames that you have probably seen when using different UI programs you have downloaded or purchased. One reason to use similar frame objects in your own programs is that users are already familiar with the frame environment. When users see frames on their computer screens, they expect to see a title bar at the top containing text information (such as "First frame"). Users also expect to see Minimize, Maximize or Restore, and Close buttons in the frame's upper-right corner. Most users assume that they can change a frame's size by dragging its border or reposition the frame on their screen by dragging the frame's title bar to a new location. The JFrame in Figure 14-3 has all of these capabilities.

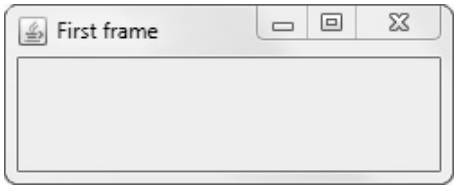

Figure 14-3 Output of the JFrame1 application

In the application in Figure 14-2, all three statements in the main() method are important. After you instantiate aFrame, if you do not use setVisible(true), you do not see the JFrame,

and if you do not set its size, you see only the title bar of the JFrame because the JFrame size is 0 × 0 by default. It might seem unusual that the default state for a JFrame is invisible. However, consider that you might want to construct a JFrame in the background while other actions are occurring and that you might want to make it visible later, when appropriate (for example, after the user has taken an action such as selecting an option). To make a frame visible, some Java programmers use the show() method instead of the setVisible() method.

When a user closes a JFrame by clicking the Close button in the upper-right corner, the default behavior is for the JFrame to become hidden and for the application to keep running. This makes sense when there are other tasks for the program to complete after the main frame is closed—for example, displaying additional frames, closing open data files, or printing an activity report. However, when a JFrame serves as a Swing application's main user interface (as happens frequently in interactive programs), you usually want the program to exit when the user clicks the Close button. To change this behavior, you can call a JFrame's setDefaultCloseOperation() method and use one of the following four values as an argument:

- JFrame.EXIT_ON_CLOSE exits the program when the JFrame is closed.
- WindowConstants.DISPOSE_ON_CLOSE closes the frame, disposes of the JFrame object, and keeps running the application.
- WindowConstants.DO_NOTHING_ON_CLOSE keeps the JFrame open and continues running. In other words, it disables the Close button.
- WindowConstants.HIDE_ON_CLOSE closes the JFrame and continues running; this is the default operation that you frequently want to override.

When you execute an application in which you have forgotten to exit when the JFrame is closed, you can end the program by typing Ctrl+C.

Each of the four usable setDefaultCloseOperation() arguments represents an integer; for example, the value of JFrame.EXIT_ON_CLOSE is 3. However, it is easier to remember the constant names than the numeric values they represent, and other programmers more easily understand your intentions if you use the named constant identifier.

Customizing a JFrame's Appearance

The appearance of the JFrame in Figure 14-3 is provided by the operating system in which the program is running (in this case, Windows). For example, the coffee-cup icon in the frame's title bar and the Minimize, Restore, and Close buttons look and act as they do in other Windows applications. The icon and buttons are known as **window decorations**; by default, window decorations are supplied by the operating system. However, you can request that Java's look and feel provide the decorations for a frame. A **look and feel** is the default appearance and behavior of any user interface.

Optionally, you can set a JFrame's look and feel using the setDefaultLookAndFeelDecorated() method. For example, Figure 14-4 shows an application that calls this method.

```
import javax.swing.*;
public class JFrame2
{
   public static void main(String[] args)
   {
      JFrame.setDefaultLookAndFeelDecorated(true);
      JFrame aFrame = new JFrame("Second frame");
      aFrame.setSize(250, 100);
      aFrame.setVisible(true);
   }
}
```

Figure 14-4 The JFrame2 class

You can provide a custom icon for a frame instead of using your operating system's default icon or the Java look-and-feel icon. For details, go to the Java Web site and search for "How to Make Frames."

The program in Figure 14-4 differs from Figure 14-2 only in the shaded areas, which show the class name, the text in the title bar, and the look-and-feel statement. Figure 14-5 shows the output. If you compare the frame in Figure 14-5 with the one in Figure 14-3, you can see that Java's look and feel has similar features to that of Windows, but their appearance is different. Java's look and feel is also known by the name *Metal*.

Figure 14-5 Output of the JFrame2 application

Look and feel is a legal issue because some software companies claim that competitors are infringing on their copyright protection by copying the look and feel of their products.

Watch the video *Using the* `JFrame` *class*.

TWO TRUTHS & A LIE

Using the JFrame Class

1. The JFrame class contains overloaded constructors; for example, you can specify a title or not.
2. An advantage of using a JFrame is that it resembles traditional frames that people are accustomed to using.
3. When a user closes a JFrame by clicking the Close button in the upper-right corner, the default behavior is for the application to end.

The false statement is #3. When a user closes a JFrame by clicking the Close button in the upper-right corner, the default behavior is for the JFrame to become hidden and for the application to keep running.

You Do It

Creating a JFrame

In this section, you create a JFrame object that appears on the screen.

1. Open a new file in your text editor, and type the following statement to import the javax.swing classes:

```
import javax.swing.*;
```

2. On the next lines, type the following class header for the JDemoFrame class, its opening curly brace, the main() method header, and its opening curly brace:

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

3. Within the body of the main() method, enter the following code to declare a JFrame with a title, set its size, and make it visible. If you neglect to set a JFrame's size, you see only the title bar of the JFrame (because the size is 0 × 0 by default); if you neglect to make the JFrame visible, you do not see anything. Add two closing curly braces—one for the main() method and one for the JDemoFrame class.

(continues)

(continued)

```
        JFrame aFrame = new JFrame("This is a frame");
        final int WIDTH = 250;
        final int HEIGHT = 250;
        aFrame.setSize(WIDTH, HEIGHT);
        aFrame.setVisible(true);
    }
}
```

4. Save the file as **JDemoFrame.java**. Compile and then run the program. The output looks like Figure 14-6—an empty `JFrame` with a title bar, a little taller than it is wide. The `JFrame` has all the properties of frames you have seen in applications you have used. For example, click the `JFrame`'s **Minimize** button, and the `JFrame` minimizes to an icon on the Windows taskbar.

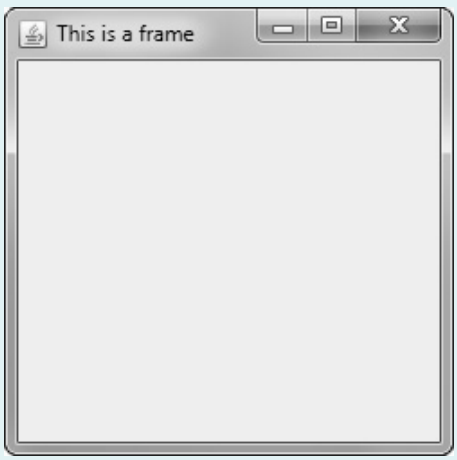

Figure 14-6 Output of the `JDemoFrame` application

5. Click the `JFrame`'s **icon** on the taskbar. The `JFrame` returns to its previous size.
6. Click the `JFrame`'s **Maximize** button. The `JFrame` fills the screen.
7. Click the `JFrame`'s **Restore** button. The `JFrame` returns to its original size.
8. Position your mouse pointer on the `JFrame`'s title bar, and then drag the `JFrame` to a new position on your screen.
9. Click the `JFrame`'s **Close** button. The `JFrame` disappears or hides. The default behavior of a `JFrame` is simply to hide when the user clicks the Close button—not to end the program.

(continues)

(continued)

10. To end the program and return control to the command line, click the Command Prompt window, and then press **Ctrl+C**. In Chapter 6, you learned to press Ctrl+C to stop a program that contains an infinite loop. This situation is similar—you want to stop a program that does not have a way to end automatically.

Ending an Application When a `JFrame` Closes

Next, you modify the `JDemoFrame` program so that the application ends when the user clicks the `JDemoFrame` Close button.

1. Within the `JDemoFrame` class file, change the class name to **`JDemoFrameThatCloses`**.
2. Add a new line of code as the final executable statement within the `main()` method, as follows:

```
aFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

3. Save the file as **JDemoFrameThatCloses.java**, and compile and execute the application.
4. When the `JFrame` appears on your screen, confirm that it still has Minimize, Maximize, and Restore capabilities. Then click the `JFrame`'s **Close** button. The `JFrame` closes, and the command prompt returns as the program relinquishes control to the operating system.

Using the `JLabel` Class

One of the components you might want to place on a `JFrame` is a `JLabel`. **`JLabel`** is a built-in Java `Swing` class that holds text you can display. The inheritance hierarchy of the `JLabel` class is shown in Figure 14-7.

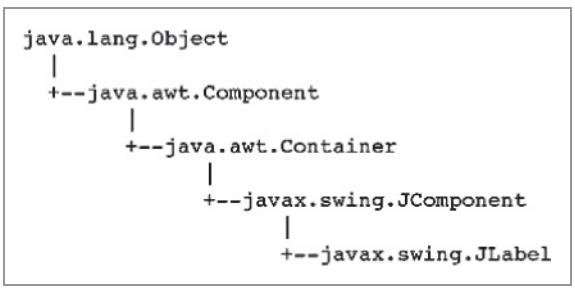

Figure 14-7 The `JLabel` class inheritance hierarchy

Available constructors for the JLabel class include the following:

- JLabel() creates a JLabel instance with no image and with an empty string for the title.
- JLabel(Icon image) creates a JLabel instance with the specified image.
- JLabel(Icon image, int horizontalAlignment) creates a JLabel instance with the specified image and horizontal alignment.
- JLabel(String text) creates a JLabel instance with the specified text.
- JLabel(String text, Icon icon, int horizontalAlignment) creates a JLabel instance with the specified text, image, and horizontal alignment.
- JLabel(String text, int horizontalAlignment) creates a JLabel instance with the specified text and horizontal alignment.

For example, you can create a JLabel named greeting that holds the words "Good day" by writing the following statement:

```
JLabel greeting = new JLabel("Good day");
```

You then can add the greeting object to the JFrame object named aFrame using the **add() method** as follows:

```
aFrame.add(greeting);
```

Figure 14-8 shows an application in which a JFrame is created and its size, visibility, and close operation are set. Then a JLabel is created and added to the JFrame. Figure 14-9 shows the output.

```
import javax.swing.*;
public class JFrame3
{
   public static void main(String[] args)
   {
      final int FRAME_WIDTH = 250;
      final int FRAME_HEIGHT = 100;
      JFrame aFrame = new JFrame("Third frame");
      aFrame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      aFrame.setVisible(true);
      aFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JLabel greeting = new JLabel("Good day");
      aFrame.add(greeting);
   }
}
```

Figure 14-8 The JFrame3 class

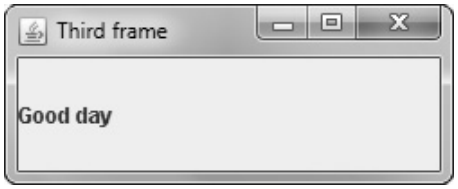

Figure 14-9 Output of the `JFrame3` application

The counterpart to the `add()` method is the **`remove()` method**. The following statement removes `greeting` from `aFrame`:

```
aFrame.remove(greeting);
```

If you add or remove a component from a container after it has been made visible, you should also call the `invalidate()`, `validate()`, and `repaint()` methods, or else you will not see the results of your actions. Each performs slightly different functions, but all three together guarantee that the results of changes in your layout will take effect. The `invalidate()` and `validate()` methods are part of the `Container` class, and the `repaint()` method is part of the `Component` class.

If you add or remove a component in a `JFrame` during construction, you do not have to call `repaint()` if you later alter the component—for example, by changing its text. You only need to call `repaint()` if you add or remove a component after construction. You will learn more about the `repaint()` method in the *Graphics* chapter.

You can change the text in a `JLabel` by using the `Component` class **`setText()` method** with the `JLabel` object and passing a `String` to it. For example, the following code changes the value displayed in the greeting `JLabel`:

```
greeting.setText("Howdy");
```

You can retrieve the text in a `JLabel` (or other `Component`) by using the **`getText()` method**, which returns the currently stored `String`.

Changing a `JLabel`'s Font

If you use the Internet and a Web browser to visit Web sites, you probably are not very impressed with the simple application displayed in Figure 14-9. You might think that the string "Good day" is plain and lackluster. Fortunately, Java provides you with a **`Font` class** from which you can create an object that holds typeface and size information. The **`setFont()` method** requires a `Font` object argument. To construct a `Font` object, you need three arguments: typeface, style, and point size.

- The **typeface argument** to the `Font` constructor is a `String` representing a font. Common fonts have names such as Arial, Century, Monospaced, and Times New Roman. The typeface argument in the `Font` constructor is only a request; the system on which

your program runs might not have access to the requested font, and if necessary, it substitutes a default font.

- The **style argument** applies an attribute to displayed text and is one of three values: `Font.PLAIN`, `Font.BOLD`, or `Font.ITALIC`.
- The **point size argument** is an integer that represents about 1/72 of an inch. Printed text is commonly 12 points; a headline might be 30 points.

In printing, point size defines a measurement between lines of text in a single-spaced text document. The point size is based on typographic points, which are approximately 1/72 of an inch. Java adopts the convention that one point on a display is equivalent to one unit in user coordinates. For more information, see the `Font` documentation at the Java Web site.

To give a `JLabel` object a new font, you can create a `Font` object, as in the following:

```
Font headlineFont = new Font("Monospaced", Font.BOLD, 36);
```

The typeface name is a `String`, so you must enclose it in double quotation marks.

You can use the `setFont()` method to assign the `Font` to a `JLabel` with a statement such as:

```
greeting.setFont(headlineFont);
```

Figure 14-10 shows a class named `JFrame4`. All the changes from `JFrame3` are shaded.

```
import javax.swing.*;
import java.awt.*;
public class JFrame4
{
   public static void main(String[] args)
   {
      final int FRAME_WIDTH = 250;
      final int FRAME_HEIGHT = 100;
      Font headlineFont = new Font("Arial", Font.BOLD, 36);
      JFrame aFrame = new JFrame("Fourth frame");
      aFrame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      aFrame.setVisible(true);
      aFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JLabel greeting = new JLabel("Good day");
      greeting.setFont(headlineFont);
      aFrame.add(greeting);
   }
}
```

Figure 14-10 The `JFrame4` program

The program in Figure 14-10 includes a new `import` statement for the package that contains the `Font` class. The program contains a `Font` object named `headlineFont` that is applied to

the greeting. Figure 14-11 shows the execution of the JFrame4 program; the greeting appears in a 36-point, bold, Arial font.

Figure 14-11 Output of the JFrame4 program

You are not required to provide an identifier for a Font. For example, you could omit the shaded statement that declares headlineFont in Figure 14-10 and set the greeting Font with the following statement that uses an anonymous Font object:

```
greeting.setFont(new Font("Arial", Font.BOLD, 36));
```

After you create a Font object, you can create a new object with a different type and size using the deriveFont() method with appropriate arguments. For example, the following two statements create a headlineFont object and a textBodyFont object that is based on the first object:

```
Font headlineFont = new Font("Arial", Font.BOLD, 36);
Font textBodyFont = headlineFont.deriveFont(Font.PLAIN, 14);
```

TWO TRUTHS & A LIE

Using the JLabel Class

1. JLabel is a built-in Java Swing class that holds text you can display.
2. You can change a JLabel's text by using its JFrame's name, a dot, and the add() method, and then using the desired text as the argument to the method.
3. If you add or remove a component from a container after it has been made visible, you should also call the validate() and repaint() methods, or else you will not see the results of your actions.

The false statement is #2. You change a JLabel's text using the setText() method, including the new text as the argument. You add a JLabel to a JFrame by using the JFrame's name, a dot, and the add() method, and then by using the JLabel's name as an argument to the method.

Using a Layout Manager

When you want to add multiple components to a `JFrame` or other container, you usually need to provide instructions for the layout of the components. For example, Figure 14-12 shows an application in which two `JLabel`s are created and added to a `JFrame` in the final shaded statements.


```
import javax.swing.*;
import java.awt.*;
public class JFrame5
{
   public static void main(String[] args)
   {
      final int FRAME_WIDTH = 250;
      final int FRAME_HEIGHT = 100;
      JFrame aFrame = new JFrame("Fifth frame");
      aFrame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      aFrame.setVisible(true);
      aFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JLabel greeting = new JLabel("Hello");
      JLabel greeting2 = new JLabel("Who are you?");
      aFrame.add(greeting);
      aFrame.add(greeting2);
   }
}
```

Figure 14-12 The `JFrame5` program

Figure 14-13 shows the output of the application in Figure 14-12. Although two `JLabel`s are added to the frame, only the last one added is visible. The second `JLabel` has been placed on top of the first one, totally obscuring it. If you continued to add more `JLabel`s to the program, only the last one added to the `JFrame` would be visible.

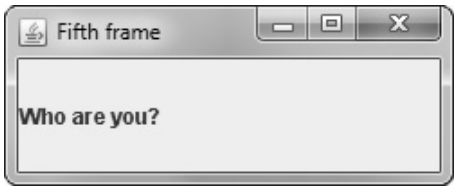

Figure 14-13 Output of the `JFrame5` program

To place multiple components at specified positions in a container so they do not hide each other, you must explicitly use a **layout manager**—a class that controls component positioning. The normal (default) behavior of a `JFrame` is to use a layout format named `BorderLayout`. A **`BorderLayout`**, created by using the `BorderLayout` class, divides a container into regions. When you do not specify a region in which to place a component (as the

JFrame5 program fails to do), all the components are placed in the same region, and they obscure each other.

When you use a FlowLayout instead of a BorderLayout, components do not lie on top of each other. Instead, the **flow layout manager** places components in a row, and when a row is filled, components automatically spill into the next row.

Three constants are defined in the FlowLayout class that specify how components are positioned in each row of their container. These constants are FlowLayout.LEFT, FlowLayout.RIGHT, and FlowLayout.CENTER. For example, to create a layout manager named flow that positions components to the right, you can use the following statement:

```
FlowLayout flow = new FlowLayout(FlowLayout.RIGHT);
```

If you do not specify how components are laid out, by default they are centered in each row.

Suppose that you create a FlowLayout object named flow as follows:

```
FlowLayout flow = new FlowLayout();
```

Then the layout of a JFrame named aFrame can be set to the newly created FlowLayout using the statement:

```
aFrame.setLayout(flow);
```

A more compact syntax that uses an anonymous FlowLayout object is:

```
aFrame.setLayout(new FlowLayout());
```

Figure 14-14 shows an application in which the JFrame's layout manager has been set so that multiple components are visible.

```
import javax.swing.*;
import java.awt.*;
public class JFrame6
{
   public static void main(String[] args)
   {
      final int FRAME_WIDTH = 250;
      final int FRAME_HEIGHT = 100;
      JFrame aFrame = new JFrame("Sixth frame");
      aFrame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      aFrame.setVisible(true);
      aFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JLabel greeting = new JLabel("Hello");
      JLabel greeting2 = new JLabel("Who are you?");
      aFrame.setLayout(new FlowLayout());
      aFrame.add(greeting);
      aFrame.add(greeting2);
   }
}
```

Figure 14-14 The JFrame6 program

Figure 14-15 shows the execution of the `JFrame6` program. Because a `FlowLayout` is used, the two `JLabel`s appear side by side. If there were more `JLabel`s or other components, they would continue to be placed side by side across the `JFrame` until there was no more room. Then, the additional components would be placed in a new row beneath the first row of components.

Figure 14-15 Output of the `JFrame6` program

Other layout managers allow you to position components in a container more precisely. You will learn about these in the *Graphics* chapter. The examples in this chapter will use `FlowLayout` because it is the easiest of the layout managers to use.

Watch the video *Using a Layout Manager*.

TWO TRUTHS & A LIE

Using a Layout Manager

1. If you do not provide a layout manager for a `JFrame`, you cannot add multiple components to it.
2. The normal (default) behavior of a `JFrame` is to use a layout format named `BorderLayout`.
3. The flow layout manager places components in a row, and when a row is filled, it automatically spills components into the next row.

The false statement is #1. If you do not provide a layout manager for a `JFrame`, you can add multiple components to it, but only the most recently added one is visible.

Extending the JFrame Class

You can instantiate a simple JFrame object within an application's main() method or with any other method of any class you write. Alternatively, you can create your own class that descends from the JFrame class. The advantage of creating a child class of JFrame is that you can set the JFrame's properties within your object's constructor; then, when you create your JFrame child object, it is automatically endowed with the features you have specified, such as title, size, and default close operation.

You already know that you create a child class by using the keyword extends in the class header, followed by the parent class name. You also know that you can call the parent class's constructor using the keyword super, and that when you call super(), the call must be the first statement in the constructor. For example, the JMyFrame class in Figure 14-16 extends JFrame. Within the JMyFrame constructor, the super() JFrame constructor is called; it accepts a String argument to use as the JFrame's title. (Alternatively, the setTitle() method could have been used.) The JMyFrame constructor also sets the size, visibility, and default close operation for every JMyFrame. Each of the methods—setSize(), setVisible(), and setDefaultCloseOperation()—appears in the constructor in Figure 14-16 without an object, because the object is the current JMyFrame being constructed. Each of the three methods could be preceded with a this reference with exactly the same meaning. That is, within the JMyFrame constructor, the following two statements have identical meanings:

```
setSize(WIDTH, HEIGHT);
this.setSize(WIDTH, HEIGHT);
```

Each statement sets the size of "this" current JMyFrame instance.

```
import javax.swing.*;
public class JMyFrame extends JFrame
{
   final int WIDTH = 200;
   final int HEIGHT = 120;
   public JMyFrame()
   {
      super("My frame");
      setSize(WIDTH, HEIGHT);
      setVisible(true);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

Figure 14-16 The JMyFrame class

Figure 14-17 shows an application that declares two JMyFrame objects. Each has the same set of attributes, determined by the JMyFrame constructor.

```
public class CreateTwoJMyFrameObjects
{
   public static void main(String[] args)
   {
      JMyFrame myFrame = new JMyFrame();
      JMyFrame mySecondFrame = new JMyFrame();
   }
}
```

Figure 14-17 The `CreateTwoJMyFrameObjects` application

When you execute the application in Figure 14-17, the two `JMyFrame` objects are displayed with the second one on top of, or obscuring, the first. Figure 14-18 shows the output of the `CreateTwoJMyFrameObjects` application after the top `JMyFrame` has been dragged to partially expose the bottom one.

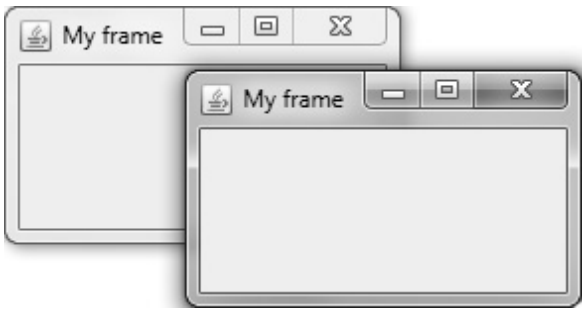

Figure 14-18 Output of the `CreateTwoJMyFrameObjects` application after dragging the top frame

You could use the `setBounds()` method with one of the `JMyFrame` objects that produces the output in Figure 14-18 so that you don't have to move one `JMyFrame` object to view the other. See Table 14-1 for details. The `Object` class also includes a `setLocation()` method you can use with a `JFrame`. To use this method, you provide horizontal and vertical position values as method arguments. You will learn more about the `setLocation()` method in the chapter *Applets, Images, and Sound*.

You exit the application when you click the Close button on either of the two `JMyFrame` objects shown in Figure 14-18. Each object has the same default close operation because each uses the same constructor that specifies this operation. To allow only one `JMyFrame` to control the program's exit, you could use the `setDefaultCloseOperation()` method with one or both of the objects in the application to change its close behavior. For example, you could use `DISPOSE_ON_CLOSE` to dismiss one of the frames but keep the application running.

When you extend a `JFrame` to create a new custom class, you must remember to make decisions as to which attributes you want to set within the class and which you want to leave to the applications that will use the class. For example, you can place the `setVisible()`

statement within the JFrame child class constructor (using either an explicit or implied this reference), or you can allow the application to use a setVisible() statement (using the name of an instantiated object followed by a dot and the method name). Either one works, but if you fail to do either, the frame will not be visible.

Programmers frequently place a main() method within a class such as JMyFrame. Then the class provides the option to be used to instantiate objects, as in the CreateTwoJMyFrameObjects application, or to be used to execute as a program that creates an object.

TWO TRUTHS & A LIE

Extending the JFrame Class

1. The advantage of creating a child class of JFrame is that you can set the JFrame's properties within your object's constructor so it is automatically endowed with the features that you have specified.
2. When a class descends from JFrame, you can use super() or setTitle() to set the title within any of the child's methods.
3. When you extend a JFrame to create a new custom class, you can decide which attributes you want to set within the class and which you want to leave to the applications that will use the class.

The false statement is #2. When a class descends from JFrame, you can use super() or setTitle() to set the title within the child's constructor. However, super() does not work in other methods.

Adding JTextFields, JButtons, and Tool Tips to a JFrame

In addition to including JLabel objects, JFrames often contain other window features, such as JTextFields, JButtons, and tool tips.

Adding JTextFields

A **JTextField** is a component into which a user can type a single line of text data. (Text data comprises any characters you can enter from the keyboard, including numbers and punctuation.) Figure 14-19 shows the inheritance hierarchy of the JTextField class.

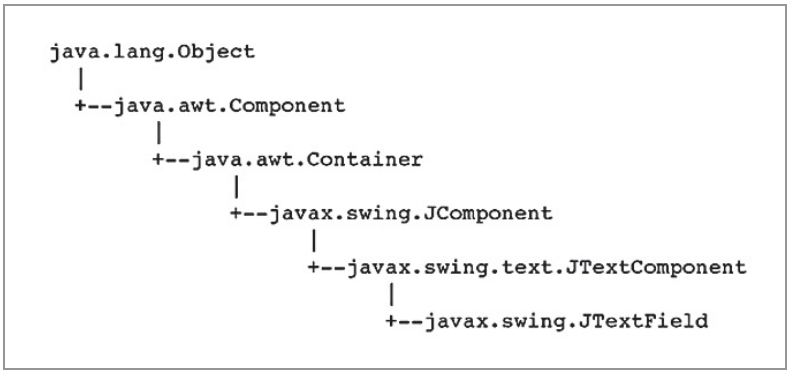

Figure 14-19 The JTextField class inheritance hierarchy

Typically, a user types a line into a JTextField and then presses Enter on the keyboard or clicks a button with the mouse to enter the data. You can construct a JTextField object using one of several constructors:

- public JTextField() constructs a new JTextField.
- public JTextField(int columns) constructs a new, empty JTextField with a specified number of columns.
- public JTextField(String text) constructs a new JTextField initialized with the specified text.
- public JTextField(String text, int columns) constructs a new JTextField initialized with the specified text and columns.

For example, to provide a JTextField that allows enough room for a user to enter approximately 10 characters, you can code the following:

```
JTextField response = new JTextField(10);
```

To add the JTextField named response to a JFrame named frame, you write:

```
frame.add(response);
```

The number of characters a JTextField can display depends on the font being used and the actual characters typed. For example, in most fonts, *w* is wider than *i*, so a JTextField of size 10 using the Arial font can display 24 *i* characters, but only eight *w* characters.

Try to anticipate how many characters your users might enter when you create a JTextField. The user can enter more characters than those that display, but the extra characters scroll out of view. It can be disconcerting to try to enter data into a field that is not large enough. It is usually better to overestimate than underestimate the size of a text field.

Several other methods are available for use with JTextFields. The setText() method allows you to change the text in a JTextField (or other Component) that has already been created, as in the following:

```
response.setText("Thank you");
```

After a user has entered text in a JTextField, you can clear it out with a statement such as the following, which assigns an empty string to the text:

```
response.setText("");
```


The getText() method allows you to retrieve the String of text in a JTextField (or other Component), as in:

```
String whatUserTyped = response.getText();
```

When a JTextField has the capability of accepting keystrokes, the JTextField is **editable**. A JTextField is editable by default. If you do not want the user to be able to enter data in a JTextField, you can send a boolean value to the **setEditable() method** to change the JTextField's editable status. For example, if you want to give a user a limited number of chances to answer a question correctly, you can count data entry attempts and then prevent the user from replacing or editing the characters in the JTextField by using a statement similar to the following:

```
if(attempts > LIMIT)
   response.setEditable(false);
```

Adding JButtons

A **JButton** is a Component the user can click with a mouse to make a selection. A JButton is even easier to create than a JTextField. There are five JButton constructors:

- public JButton() creates a button with no set text.
- public JButton(Icon icon) creates a button with an icon of type Icon or ImageIcon.
- public JButton(String text) creates a button with text.
- public JButton(String text, Icon icon) creates a button with initial text and an icon of type Icon or ImageIcon.
- public JButton(Action a) creates a button in which properties are taken from the Action supplied. (Action is a Java class.)

The inheritance hierarchy of the JButton class is shown in Figure 14-20.

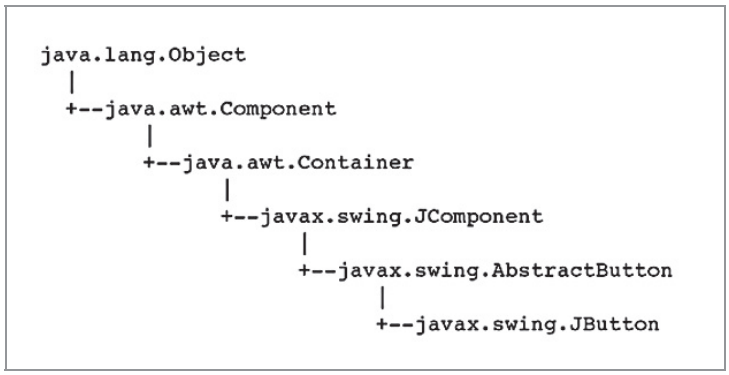

Figure 14-20 The JButton class inheritance hierarchy

To create a JButton with the label "Press when ready", you can write the following:

```
JButton readyJButton = new JButton("Press when ready");
```

You can add a JButton to a JFrame (or other container) using the add() method. You can change a JButton's label with the setText() method, as in:

```
readyJButton.setText("Don't press me again!");
```

You can retrieve the text from a JButton and assign it to a String object with the getText() method, as in:

```
String whatsOnJButton = readyJButton.getText();
```

Figure 14-21 shows a class that extends JFrame and holds several components. As the components (two JLabels, a JTextField, and a JButton) are added to the JFrame, they are placed from left to right in horizontal rows across the JFrame's surface. Figure 14-22 shows the program that instantiates an instance of the JFrame.

```
import javax.swing.*;
import java.awt.*;
public class JFrameWithManyComponents extends JFrame
{
   final int FRAME_WIDTH = 300;
   final int FRAME_HEIGHT = 150;
   public JFrameWithManyComponents()
   {
      super("Demonstrating many components");
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JLabel heading = new JLabel("This frame has many components");
      heading.setFont(new Font("Arial", Font.BOLD, 16));
      JLabel namePrompt = new JLabel("Enter your name:");
      JTextField nameField = new JTextField(12);
      JButton button = new JButton("Click to continue");
      setLayout(new FlowLayout());
      add(heading);
      add(namePrompt);
      add(nameField);
      add(button);
   }
}
```

Figure 14-21 The JFrameWithManyComponents class

```
public class ComponentDemo
{
   public static void main(String[] args)
   {
      JFrameWithManyComponents frame =
         new JFrameWithManyComponents();
      frame.setVisible(true);
   }
}
```

Figure 14-22 A ComponentDemo application that instantiates a JFrameWithManyComponents

When you execute the ComponentDemo program, the JFrame contains all the components that were added in the frame's constructor, as shown in Figure 14-23. A user can minimize or restore the frame and can alter its size by dragging the frame borders. The user can type characters in the JTextField and click the JButton. When the button is clicked, it appears to be pressed just like buttons you have used in professional applications. However, when the user types characters or clicks the button, no resulting actions occur because code has not yet been written to handle those user-initiated events.

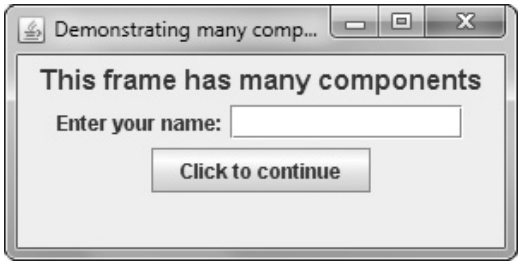

Figure 14-23 Execution of the ComponentDemo program

Using Tool Tips

Tool tips are popup windows that can help a user understand the purpose of components in an application; the tool tip appears when a user hovers the mouse pointer over the component. You define the text to be displayed in a tool tip by using the **setToolTipText() method** and passing an appropriate String to it. For example, in the JFrameWithManyComponents program in Figure 14-21, you can add a tool tip to the button component by using the following statement in the JFrame constructor:

```
button.setToolTipText("Click this button");
```

Figure 14-24 shows the result when the JFrame is displayed and the user's mouse pointer is placed over the button.

Figure 14-24 JFrame with added tool tip

The JFrameWithToolTip.java file in your downloadable student files contains a revised version of JFrameWithManyComponents with the tool tip added. The ToolTipDemo.java file contains an application that instantiates a JFrameWithToolTip object.

TWO TRUTHS & A LIE

Adding JTextFields, JButtons, and Tool Tips to a JFrame

1. A JTextField is a component into which a user can type a single line of text data; typically, a user types a line into a JTextField and then presses Enter on the keyboard or clicks a button with the mouse to enter the data.
2. A JButton is a Component the user can click with a mouse to make a selection.
3. Tool tips are the different symbols you can select to display as a cursor in your applications.

The false statement is #3. Tool tips are popup windows that can help a user understand the purpose of components in an application; the tool tip appears when a user hovers the mouse pointer over the component.

You Do It

Adding Components to a JFrame

Next, you create a Swing application that displays a JFrame that holds a JLabel, JTextField, and JButton.

1. Open a new file in your text editor, then type the following first few lines of an application. The import statements make the Swing and AWT components available, and the class header indicates that the class is a JFrame. The class contains several components: a label, field, and button.

```
import javax.swing.*;
import java.awt.*;
public class JFrameWithComponents extends JFrame
{
    JLabel label = new JLabel("Enter your name");
    JTextField field = new JTextField(12);
    JButton button = new JButton("OK");
```

2. In the JFrameWithComponents constructor, set the JFrame title to "Frame with Components" and the default close operation to exit the program when the JFrame is closed. Set the layout manager. Add the label, field, and button to the JFrame.

```
public JFrameWithComponents()
{
    super("Frame with Components");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new FlowLayout());
    add(label);
    add(field);
    add(button);
}
```

3. Add a closing curly brace for the class, and then save the file as **JFrameWithComponents.java**.
4. Compile the class and correct any errors.
5. Next, write an application that creates a new JFrameWithComponents named aFrame, sizes it using the setSize() method, and then sets its visible property to true.

(continues)

(continued)

```
import javax.swing.*;
public class CreateJFrameWithComponents
{
   public static void main(String[] args)
   {
      JFrameWithComponents aFrame =
          new JFrameWithComponents();
      final int WIDTH = 350;
      final int HEIGHT = 100;
      aFrame.setSize(WIDTH, HEIGHT);
      aFrame.setVisible(true);
   }
}
```

6. Save the file as **CreateJFrameWithComponents.java**. Compile and then execute the application. The output is shown in Figure 14-25.

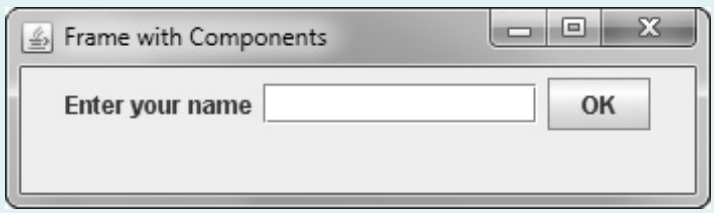

Figure 14-25 Output of the `CreateJFrameWithComponents` application

7. Click the `JButton`. It acts like a button should—that is, it appears to be pressed when you click it, but nothing happens because you have not yet written instructions for the button clicks to execute.
8. Close the application.

Learning About Event-Driven Programming

An **event** occurs when a user takes action on a component, such as clicking the mouse on a `JButton` object. In an **event-driven program**, the user might initiate any number of events in any order. For example, if you use a word-processing program, you have dozens of choices at your disposal at any time. You can type words, select text with the mouse, click a button to change text to bold, click a button to change text to italic, choose a menu item, and so on. With each word-processing document you create, you choose options in any order that seems appropriate at the time. The word-processing program must be ready to respond to any event you initiate.

Within an event-driven program, a component on which an event is generated is the **source** of the event. A button that a user can click is an example of a source; a text field that a user can use to enter text is another source. An object that is interested in an event is a **listener**. Not all objects listen for all possible events—you probably have used programs in which clicking many areas of the screen has no effect. If you want an object to be a listener for an event, you must register the object as a listener for the source.

Social networking sites maintain lists of people in whom you are interested and notify you each time a person on your list posts a comment or picture. Similarly, a Java component source object (such as a button) maintains a list of registered listeners and notifies all of them when any event occurs. For example, a `JFrame` might want to be notified of any mouse click on its surface. When the listener "receives the news," an event-handling method contained in the listener object responds to the event.

A source object and a listener object can be the same object. For example, you might program a `JButton` to change its own label when a user clicks it.

To respond to user events within any class you create, you must do the following:

- Prepare your class to accept event messages.
- Tell your class to expect events to happen.
- Tell your class how to respond to events.

Preparing Your Class to Accept Event Messages

You prepare your class to accept button-press events by importing the `java.awt.event` package into your program and adding the phrase `implements ActionListener` to the class header. The `java.awt.event` package includes event classes with names such as `ActionEvent`, `ComponentEvent`, and `TextEvent`. `ActionListener` is an interface—a class containing a set of specifications for methods that you can use. Implementing `ActionListener` provides you with standard event method specifications that allow your listener to work with `ActionEvents`, which are the types of events that occur when a user clicks a button.

You learned to create and implement interfaces in Chapter 11. You can identify interfaces such as `ActionListener` because they use the keyword `implements`. In ordinary language, an item that is implemented is put into service, or used. Implementation has a similar meaning when applied to interfaces. In contrast, packages that are imported are brought into an application, and classes that are added onto are extended.

Telling Your Class to Expect Events to Happen

You tell your class to expect `ActionEvents` with the **`addActionListener()` method**. If you have declared a `JButton` named `aButton`, and you want to perform an action when a user clicks `aButton`, `aButton` is the source of a message, and you can think of your class as a target to which to send a message. You learned in Chapter 4 that the `this` reference means "this current object," so the code `aButton.addActionListener(this);` causes any `ActionEvent` messages (button clicks) that come from `aButton` to be sent to "this current object."

Not all `Events` are `ActionEvents` with an `addActionListener()` method. For example, `KeyListeners` have an `addKeyListener()` method, and `FocusListeners` have an `addFocusListener()` method. Additional event types and methods are covered in more detail in the next chapter.

Telling Your Class How to Respond to Events

The `ActionListener` interface contains the **`actionPerformed(ActionEvent e)` method** specification. When a class, such as a `JFrame`, has registered as a listener with a `Component` such as a `JButton`, and a user clicks the `JButton`, the `actionPerformed()` method executes. You implement the `actionPerformed()` method, which contains a header and a body, like all methods. You use the following header, in which `e` represents any name you choose for the `Event` (the `JButton` click) that initiated the notification of the `ActionListener` (which is the `JFrame`):

```
public void actionPerformed(ActionEvent e)
```

The body of the method contains any statements that you want to execute when the action occurs. You might want to perform mathematical calculations, construct new objects, produce output, or execute any other operation. For example, Figure 14-26 shows a `JFrame` containing a `JLabel` that prompts the user for a name, a `JTextField` into which the user can type a response, a `JButton` to click, and a second `JLabel` that displays the name entered by the user. The `actionPerformed()` method executes when the user clicks the `pressMe JButton`; within the method, the `String` that a user has typed into the `JTextField` is retrieved and stored in the `name` variable. The name is then used as part of a `String` that alters the second `JLabel` on the `JFrame`. Figure 14-27 shows an application that instantiates a `JHelloFrame` object and makes it visible.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JHelloFrame extends JFrame implements ActionListener
{
   JLabel question = new JLabel("What is your name?");
   Font bigFont = new Font("Arial", Font.BOLD, 16);
   JTextField answer = new JTextField(10);
   JButton pressMe = new JButton("Press me");
   JLabel greeting = new JLabel("");
   final int WIDTH = 275;
   final int HEIGHT = 225;
   public JHelloFrame()
   {
      super("Hello Frame");
      setSize(WIDTH, HEIGHT);
      setLayout(new FlowLayout());
      question.setFont(bigFont);
      greeting.setFont(bigFont);
      add(question);
      add(answer);
      add(pressMe);
      add(greeting);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      pressMe.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
       String name = answer.getText();
       String greet = "Hello, " + name;
       greeting.setText(greet);
    }
}
```

Figure 14-26 The `JHelloFrame` class that produces output when the user clicks the `JButton`

```
public class JHelloDemo
{
   public static void main(String[] args)
   {
      JHelloFrame frame = new JHelloFrame();
      frame.setVisible(true);
   }
}
```

Figure 14-27 An application that instantiates a `JHelloFrame` object

Figure 14-28 shows a typical execution of the `JHelloDemo` program. The user enters *Lindsey* into the `JTextField`, and the greeting with the name is displayed after the user clicks the button.

Figure 14-28 Typical execution of the `JHelloDemo` program

When more than one component is added and registered to a `JFrame`, it might be necessary to determine which component was used to initiate an event. For example, in the `JHelloFrame` class in Figure 14-26, you might want the user to be able to see the message after either clicking the button or pressing Enter in the `JTextField`. In that case, you would designate both the `pressMe` button and the `answer` text field to be message sources by using the `addActionListener()` method with each, as follows:

```
pressMe.addActionListener(this);
answer.addActionListener(this);
```

These two statements make the `JFrame (this)` the receiver of messages from either object. The `JFrame` has only one `actionPerformed()` method, so it is the method that executes when either the `pressMe` button or the `answer` text field sends a message.

If you want different actions to occur depending on whether the user clicks the button or presses Enter, you must determine the source of the event. Within the `actionPerformed()` method, you can use the `getSource()` method of the object sent to determine which component generated the event. For example, within a method with the header `public void actionPerformed(ActionEvent e)`, `e` is an `ActionEvent`. `ActionEvent` and other event classes are part of the `java.awt.event` package and are subclasses of the `EventObject` class. To determine what object generated the `ActionEvent`, you can use the following statement:

```
Object source = e.getSource();
```

For example, if a `JFrame` contains two `JButtons` named `option1` and `option2`, you can use the decision structure in the method in Figure 14-29 to take different courses of action based on the button that is clicked. Whether an event's source is a `JButton`, `JTextField`, or other `Component`, it can be assigned to an `Object` because all components descend from `Object`.

```
public void actionPerformed(ActionEvent e)
{
   Object source = e.getSource();
   if(source == option1)
      //execute these statements when user clicks option1
   else
      //execute these statements when user clicks any other option
}
```

Figure 14-29 An `actionPerformed()` method that takes one of two possible actions

Alternatively, you can also use the `instanceof` keyword to determine the source of the event. The `instanceof` keyword is used when it is necessary to know only the component's type, rather than what component triggered the event. For example, if you want to take some action when a user enters data into any `JTextField`, but not when an event is generated by a different `Component` type, you could use the method format shown in Figure 14-30.

```
void actionPerformed(ActionEvent e)
{
   Object source = e.getSource();
   if(source instanceof JTextField)
   {
     // execute these statements when any JTextField
     // generates the event
     // but not when a JButton or other Component does
   }
}
```

Figure 14-30 An `actionPerformed()` method that executes a block of statements when a user generates an event from any `JTextField`

Using the `setEnabled()` Method

You probably have used computer programs in which a component becomes disabled or unusable. For example, a `JButton` might become dim and unresponsive when the programmer no longer wants you to have access to the `JButton`'s functionality. Components are enabled by default, but you can use the **`setEnabled()` method** to make a component available or unavailable by passing `true` or `false` to it, respectively. For example, Figure 14-31 shows a `JFrame` with two `JButton` objects. The one on top is enabled, but the one on the bottom has been disabled.

Figure 14-31 A `JFrame` with an enabled `JButton` and a disabled `JButton`

Your downloadable student files contain a file named JTwoButtons.java that produces the `JFrame` shown in Figure 14-31.

TWO TRUTHS & A LIE

Learning About Event-Driven Programming

1. Within an event-driven program, a component on which an event is generated is a listener.
2. You prepare your class to accept button-press events by importing the `java.awt.event` package into your program and adding the phrase `implements ActionListener` to the class header.
3. A class that can react to `ActionEvents` includes an `actionPerformed()` method.

The false statement is #1. Within an event-driven program, a component on which an event is generated is the source of the event, and an object that is interested in an event is a listener.

You Do It

Adding Functionality to a JButton and a JTextField

Next, you add functionality to the `JButton` and `JTextField` that you created in the `JFrameWithComponents` class.

1. Open the **JFrameWithComponents.java** file. Immediately save the file as **JAction.java**.
2. After the existing import statements at the top of the file, add the following import statement that will allow event handling:

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

3. Change the class name to `JAction` to match the new filename. Also change the constructor header to match the new class name. Within the constructor, change the string argument to the `super()` method from "Frame with Components" to "Action".
4. After `extends JFrame` at the end of the `JAction` class header, add the following phrase so that the class can respond to `ActionEvent`s:

```
implements ActionListener
```

5. Register the `JAction` class as a listener for events generated by either the button or the text field by adding the following statements at the end of, but within, the `JAction()` constructor:

```
button.addActionListener(this);
field.addActionListener(this);
```

6. Just prior to the closing curly brace for the class, add the following `actionPerformed()` method. The method changes the text on both the label and the button whenever the user clicks the button or presses Enter in the text field.

```
public void actionPerformed(ActionEvent e)
{
    label.setText("Thank you");
    button.setText("Done");
}
```

7. Just after the `actionPerformed()` method, and just before the closing curly brace for the class, add a `main()` method to the class so that you can instantiate a `JAction` object for demonstration purposes.

(continues)

(continued)

```
public static void main(String[] args)
{
    JAction aFrame = new JAction();
    final int WIDTH = 250;
    final int HEIGHT = 100;
    aFrame.setSize(WIDTH, HEIGHT);
    aFrame.setVisible(true);
}
```

8. Save the file, then compile and execute it. The output looks like the frame on the left side of Figure 14-32. Type a name in the text field, and then click the **OK** button. Its text changes to "Done", and its size increases slightly because the label "Done" requires more space than the label "OK". The other label requires less space than it did because "Thank you" is a shorter message than "Enter your name". Therefore, all the components are redistributed because the `FlowLayout` manager places as many components as will fit horizontally in the top row before adding components to subsequent rows. The output looks like the right side of Figure 14-32.

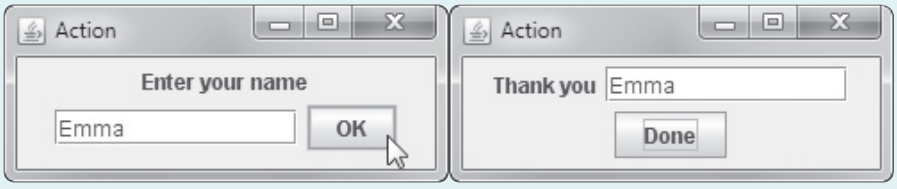

Figure 14-32 Typical execution of the `JAction` application after the user clicks the OK button

9. Close the application and then execute it again. This time, enter a name in the text field and press **Enter**. Again, the button text changes, showing that the `actionPerformed()` method reacts to actions that take place on either the button or the text field.
10. Close the application.

Distinguishing Event Sources

Next, you will modify the `actionPerformed()` method of the `JAction` class so that different results occur depending on which action a user takes.

1. Open the **JAction.java** file in your text editor if the file is not still open. Immediately save the file as **JAction2.java**.

(continues)

(continued)

2. Change the class name and the constructor name to match the new filename by adding **2** to each name.
3. In the `main()` method, change the statement that instantiates the `JFrame` object to the following:

```
JAction2 aFrame = new JAction2();
```

4. Within the `actionPerformed()` method, you can use the named `ActionEvent` argument and the `getSource()` method to determine the source of the event. Using an `if` statement, you can take different actions when the argument represents different sources. For example, you can change the label in the frame to indicate the event's source. Change the `actionPerformed()` method to:

```
public void actionPerformed(ActionEvent e)
{
   Object source = e.getSource();
   if(source == button)
      label.setText("You clicked the button");
   else
      label.setText("You pressed Enter");
}
```

5. Save the file (as JAction2.java), then compile and execute it. Type a name, press **Enter** or click the button, and notice the varying results in the frame's label. For example, Figure 14-33 shows the application after the user has typed a name and pressed Enter.

Figure 14-33 Typical execution of the `JAction2` application

6. Close the application.

Understanding `Swing` Event Listeners

Many types of listeners exist in Java, and each of these listeners can handle a specific event type. A class can implement as many event listeners as it needs—for example, a class might need to respond to both a mouse button press and a keyboard key press, so you might

implement both ActionListener and KeyListener interfaces. Table 14-2 lists some event listeners and the types of events for which they are used.

Listener	Type of Events	Example
ActionListener	Action events	Button clicks
AdjustmentListener	Adjustment events	Scroll bar moves
ChangeListener	Change events	Slider is repositioned
FocusListener	Keyboard focus events	Text field gains or loses focus
ItemListener	Item events	Check box changes status
KeyListener	Keyboard events	Text is entered
MouseListener	Mouse events	Mouse clicks
MouseMotionListener	Mouse movement events	Mouse rolls
WindowListener	Window events	Window closes

Table 14-2 Alphabetical list of some event listeners

An event occurs every time a user types a character or clicks a mouse button. Any object can be notified of an event as long as it implements the appropriate interface and is registered as an event listener on the appropriate event source. You already know that you establish a relationship between a JButton and a JFrame that contains it by using the addActionListener() method. Similarly, you can create relationships between other Swing components and the classes that react to users' manipulations of them. In Table 14-3, each component listed on the left is associated with a method on the right. For example, when you want a JCheckBox to respond to a user's clicks, you can use the addItemListener() method to register the JCheckBox as the type of object that can create an ItemEvent. The argument you place within the parentheses of the call to the addItemListener() method is the object that should respond to the event—perhaps a JFrame that contains the event-generating JCheckBox. The format is:

```
theSourceOfTheEvent.addListenerMethod(theClassThatShouldRespond);
```

Component(s)	Associated Listener-Registering Method(s)
JButton, JCheckBox, JComboBox, JTextField, and JRadioButton	addActionListener()
JScrollBar	addAdjustmentListener()
All Swing components	addFocusListener(), addKeyListener(), addMouseListener(), and addMouseMotionListener()
JButton, JCheckBox, JComboBox, and JRadioButton	addItemListener()
All JWindow and JFrame components	addWindowListener()
JSlider and JCheckBox	addChangeListener()

Table 14-3 Some Swing components and their associated listener-registering methods

Any event source can have multiple listeners registered on it. Conversely, a single listener can be registered with multiple event sources. In other words, a single instance of JCheckBox might generate ItemEvents and FocusEvents, and a single instance of the JFrame class might respond to ActionEvents generated by a JButton and ItemEvents generated by a JCheckBox.

The class of the object that responds to an event must contain a method that accepts the event object created by the user's action. A method that executes because it is called automatically when an appropriate event occurs is an **event handler**. In other words, when you register a component (such as a JFrame) to be a listener for events generated by another component (such as a JCheckBox), you must write an event handler method. You cannot choose your own name for event handlers—specific method identifiers react to specific event types. Table 14-4 lists just some of the methods that react to events.

Listener	Method
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
FocusListener	focusGained(FocusEvent) and focusLost(FocusEvent)
ItemListener	itemStateChanged(ItemEvent)

Table 14-4 Selected methods that respond to events

Each listener in Table 14-4 is associated with only one or two methods. Other listeners, such as KeyListener and MouseListener, are associated with multiple methods. You will learn how to use these more complicated listeners in the chapter *Advanced GUI Topics*.

Until you become familiar with the event-handling model, it can seem quite confusing. For now, remember these tasks you must perform when you declare a class that handles an event:

- The class that handles an event must either implement a listener interface or extend a class that implements a listener interface. For example, if a `JFrame` named `MyFrame` needs to respond to a user's clicks on a `JCheckBox`, you would write the following class header:

```
public class MyFrame extends JFrame
   implements ItemListener
```

  If you then declare a class that extends `MyFrame`, you need not include `implements ItemListener` in its header. The new class inherits the implementation.

- You must register each instance of the event-handling class as a listener for one or more components. For example, if `MyFrame` contains a `JCheckBox` named `myCheckBox`, then within the `MyFrame` class you would code:

```
myCheckBox.addItemListener(this);
```

  The `this` reference is to the class in which `myCheckBox` is declared—in this case, `MyFrame`.

- You must write an event handler method with an appropriate identifier (as shown in Table 14-4) that accepts the generated event and reacts to it.

Watch the video *Event-Driven Programming*.

TWO TRUTHS & A LIE

Understanding Swing Event Listeners

1. A class can implement as many event listeners as it needs.
2. Any object can be notified of a mouse click or keyboard press as long as it implements the appropriate interface and is registered as an event listener on the appropriate event source.
3. Every event-handling method accepts a parameter that represents the listener for the event.

The false statement is #3. Every event-handling method accepts a parameter that represents the generated event.

Using the JCheckBox, ButtonGroup, and JComboBox Classes

Besides JButtons and JTextFields, several other Java components allow a user to make selections in a UI environment. These include JCheckBoxes, ButtonGroups, and JComboBoxes.

The JCheckBox Class

A **JCheckBox** consists of a label positioned beside a square; you can click the square to display or remove a check mark. Usually, you use a JCheckBox to allow the user to turn an option on or off. For example, Figure 14-34 shows the code for an application that uses four JCheckBoxes, and Figure 14-35 shows the output.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class CheckBoxDemonstration
   extends JFrame implements ItemListener
{
   FlowLayout flow = new FlowLayout();
   JLabel label = new JLabel("What would you like to drink?");
   JCheckBox coffee = new JCheckBox("Coffee", false);
   JCheckBox cola = new JCheckBox("Cola", false);
   JCheckBox milk = new JCheckBox("Milk", false);
   JCheckBox water = new JCheckBox("Water", false);
   String output, insChosen;
   public CheckBoxDemonstration()
   {
      super("CheckBox Demonstration");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new FlowLayout());
      label.setFont(new Font("Arial", Font.ITALIC, 22));
      coffee.addItemListener(this);
      cola.addItemListener(this);
      milk.addItemListener(this);
      water.addItemListener(this);
      add(label);
      add(coffee);
      add(cola);
      add(milk);
      add(water);
  }
```

Figure 14-34 The CheckBoxDemonstration class *(continues)*

(continued)

```
   public void itemStateChanged(ItemEvent check)
   {
       // Actions based on choice go here
   }
   public static void main(String[] arguments)
   {
       final int FRAME_WIDTH = 350;
       final int FRAME_HEIGHT = 120;
       CheckBoxDemonstration frame =
          new CheckBoxDemonstration();
       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
       frame.setVisible(true);
   }
}
```

Figure 14-34 The CheckBoxDemonstration class

In the application in Figure 14-34, the CheckBoxDemonstration class and the main() method that instantiates an instance of it are part of the same class. You could also store the two parts in separate classes, as in previous examples.

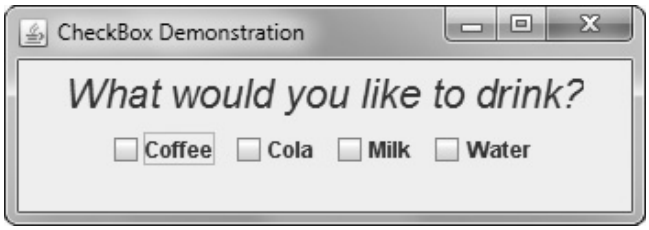

Figure 14-35 Output of the CheckBoxDemonstration class

The inheritance hierarchy of the JCheckBox class is shown in Figure 14-36; frequently used JCheckBox methods appear in Table 14-5.

```
java.lang.Object
  |--java.awt.Component
        |--java.awt.Container
              |--javax.swing.JComponent
                    |--javax.swing.AbstractButton
                          |--javax.swing.JToggleButton
                                |--javax.swing.JCheckBox
```

Figure 14-36 The inheritance hierarchy of the JCheckBox class

Method	Purpose
`void setText(String)`	Sets the text for the `JCheckBox`
`String getText()`	Returns the `JCheckBox` text
`void setSelected(boolean)`	Sets the state of the `JCheckBox` to `true` for selected or `false` for unselected
`boolean isSelected()`	Gets the current state (checked or unchecked) of the `JCheckBox`

Table 14-5 Frequently used `JCheckBox` methods

Several constructors can be used with `JCheckBox`es. When you construct a `JCheckBox`, you can choose whether to assign it a label; you can also decide whether the `JCheckBox` appears selected (`JCheckBox`es start unselected by default). The following statements create four `JCheckBox` objects—one with no label and unselected, two with labels and unselected, and one with a label and selected.

- ```
  JCheckBox box1 = new JCheckBox();
      // No label, unselected
  ```
- ```
  JCheckBox box2 = new JCheckBox("Check here");
      // Label, unselected
  ```
- ```
  JCheckBox box3 = new JCheckBox("Check here", false);
      // Label, unselected
  ```
- ```
  JCheckBox box4 = new JCheckBox("Check here", true);
      // Label, selected
  ```

If you do not initialize a `JCheckBox` with a label and you want to assign one later, or if you want to change an existing label, you can use the `setText()` method, as in the following example:

```
box1.setText("Check this box now");
```

You can set the state of a `JCheckBox` with the `setSelected()` method; for example, you can use the following statement to ensure that `box1` is unchecked:

```
box1.setSelected(false);
```

The `isSelected()` method is most useful in Boolean expressions, as in the following example, which adds one to a `voteCount` variable if `box2` is currently checked.

```
if(box2.isSelected())
   ++voteCount;
```

When the status of a `JCheckBox` changes from unchecked to checked (or from checked to unchecked), an `ItemEvent` is generated, and the `itemStateChanged()` method executes. You can use the `getItem()` method to determine which object generated the event and the `getStateChange()` method to determine whether the event was a selection or a deselection.

The `getStateChange()` method returns an integer that is equal to one of two class variables—`ItemEvent.SELECTED` or `ItemEvent.DESELECTED`. For example, in Figure 14-37 the `itemStateChanged()` method calls the `getItem()` method, which returns the object named `source`. Then, the value of `source` is tested in an `if` statement to determine if it is equivalent to a `JCheckBox` object named `checkBox`. If the two references are to the same object, the code determines whether the `checkBox` was selected or deselected, and in each case appropriate actions are taken.

```
public void itemStateChanged(ItemEvent e)
{
   Object source = e.getItem();
   if(source == checkBox)
   {
      int select = e.getStateChange();
      if(select == ItemEvent.SELECTED)
         // statements that execute when the box is checked
      else
         // statements that execute when the box is unchecked
   }
   else
   {
      // statements that execute when the source of the event is
      // some component other than the checkBox object
   }
}
```

Figure 14-37 Typical `itemStateChanged()` method

The ButtonGroup Class

Sometimes, you want options to be mutually exclusive—that is, you want the user to be able to select only one of several choices. When you create a **`ButtonGroup`**, you can group several components, such as `JCheckBox`es, so a user can select only one at a time. When you group `JCheckBox` objects, all of the other `JCheckBox`es are automatically turned off when the user selects any one check box. The inheritance hierarchy for the `ButtonGroup` class is shown in Figure 14-38. You can see that `ButtonGroup` descends directly from the `Object` class. Even though it does not begin with a *J*, the `ButtonGroup` class is part of the `javax.swing` package.

```
java.lang.Object
  |
  +--javax.swing.ButtonGroup
```

Figure 14-38 The inheritance hierarchy for the `ButtonGroup` class

A group of JCheckBoxes in which a user can select only one at a time also acts like a set of radio buttons (for example, those used to select preset radio stations on an automobile radio), which you can create using the JRadioButton class. The JRadioButton class is very similar to the JCheckBox class, and you might prefer to use it when you have a list of mutually exclusive user options. It makes sense to use ButtonGroups with items that can be selected (that is, those that use an isSelected() method). You can find more information about the JRadioButton class at *http://java.sun.com*.

To create a ButtonGroup in a JFrame and then add a JCheckBox, you must perform four steps:

- Create a ButtonGroup, such as `ButtonGroup aGroup = new ButtonGroup();`.
- Create a JCheckBox, such as `JCheckBox aBox = new JCheckBox();`.
- Add aBox to aGroup with `aGroup.add(aBox);`.
- Add aBox to the JFrame with `add(aBox);`.

You can create a ButtonGroup and then create the individual JCheckBox objects, or you can create the JCheckBoxes and then create the ButtonGroup. If you create a ButtonGroup but forget to add any JCheckBox objects to it, then the JCheckBoxes act as individual, nonexclusive check boxes.

A user can set one of the JCheckBoxes within a group to "on" by clicking it with the mouse, or the programmer can select a JCheckBox within a ButtonGroup with a statement such as the following:

```
aGroup.setSelected(aBox);
```

Only one JCheckBox can be selected within a group. If you assign the selected state to a JCheckBox within a group, any previous assignment is negated.

You can determine which, if any, of the JCheckBoxes in a ButtonGroup is selected using the isSelected() method.

After a JCheckBox in a ButtonGroup has been selected, one in the group will always be selected. In other words, you cannot "clear the slate" for all the items that are members of a ButtonGroup. You could cause all the JCheckBoxes in a ButtonGroup to initially *appear* unselected by adding one JCheckBox that is not visible (using the setVisible() method). Then, you could use the setSelected() method to select the invisible JCheckBox, and all the others would appear to be deselected.

The JComboBox Class

A **JComboBox** is a component that combines two features: a display area showing a default option and a list box that contains additional, alternate options. (A list box is also known as a combo box or a drop-down list.) The display area contains either a button that a user can click or an editable field into which the user can type. When a JComboBox appears on the screen, the default option is displayed. When the user clicks the JComboBox, a list of

alternative items drops down; if the user selects one, it replaces the box's displayed item. Users often expect to view JComboBox options in alphabetical order. If it makes sense for your application, consider displaying your options this way. Other reasonable approaches are to place choices in logical order, such as "small", "medium", and "large", or to position the most frequently selected options first.

Figure 14-39 shows a JComboBox as it looks when first displayed and as it looks after a user clicks it. The inheritance hierarchy of the JComboBox class is shown in Figure 14-40.

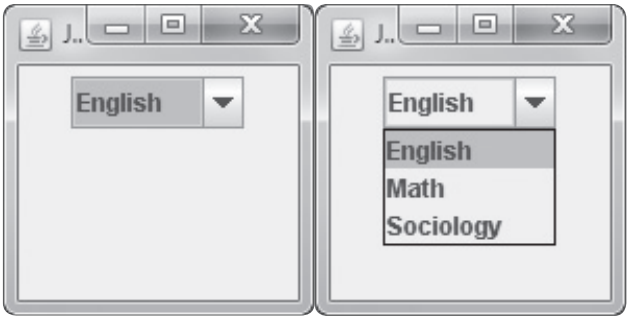

Figure 14-39 A JComboBox before and after the user clicks it

The code that produces the JComboBox in Figure 14-39 is contained in the file named ComboBoxDemonstration.java in your downloadable student files.

```
java.lang.Object
  |--java.awt.Component
        |--java.awt.Container
              |--javax.swing.JComponent
                    |--javax.swing.JComboBox
```

Figure 14-40 The inheritance hierarchy of the JComboBox class

You can build a JComboBox by using a constructor with no arguments and then adding items (for example, Strings) to the list with the addItem() method. The following statements create a JComboBox named majorChoice that contains three options from which a user can choose:

```
JComboBox<String> majorChoice = new JComboBox<String>();
majorChoice.addItem("English");
majorChoice.addItem("Math");
majorChoice.addItem("Sociology");
```

In the declaration of the JComboBox, notice the use of <String> following the class name. By default, a JComboBox expects items that are added to be Object types. Adding the angle brackets and String notifies the compiler of the expected items in the JComboBox and allows the compiler to check for errors if invalid items are added. If you do not insert a data type for a JComboBox, the program compiles, but a warning message is issued with each addItem() method call. Programmers say that JComboBox uses *generics*. **Generic programming** is a feature of modern languages that allows multiple data types to be used safely with methods.

As an alternative, you can construct a JComboBox using an array of Objects as the constructor argument; the Objects in the array become the listed items within the JComboBox. For example, the following code creates the same majorChoice JComboBox as the preceding code:

```
String[] majorArray = {"English", "Math", "Sociology"};
JComboBox majorChoice = new JComboBox(majorArray);
```

Table 14-6 lists some methods you can use with a JComboBox object. For example, you can use the setSelectedItem() or setSelectedIndex() method to choose one of the items in a JComboBox to be the initially selected item. You also can use the getSelectedItem() or getSelectedIndex() method to discover which item is currently selected.

Method	Purpose
void addItem(Object)	Adds an item to the list
void removeItem(Object)	Removes an item from the list
void removeAllItems()	Removes all items from the list
Object getItemAt(int)	Returns the list item at the index position specified by the integer argument
int getItemCount()	Returns the number of items in the list
int getMaximumRowCount()	Returns the maximum number of items the combo box can display without a scroll bar
int getSelectedIndex()	Returns the position of the currently selected item
Object getSelectedItem()	Returns the currently selected item
Object[] getSelectedObjects()	Returns an array containing selected Objects
void setEditable(boolean)	Sets the field to be editable or not editable
void setMaximumRowCount(int)	Sets the number of rows in the combo box that can be displayed at one time
void setSelectedIndex(int)	Sets the index at the position indicated by the argument
void setSelectedItem(Object)	Sets the selected item in the combo box display area to be the Object argument

Table 14-6 Some JComboBox class methods

You can treat the list of items in a JComboBox object as an array; the first item is at position 0, the second is at position 1, and so on. It is convenient to use the getSelectedIndex() method to determine the list position of the currently selected item; then you can use the index to access corresponding information stored in a parallel array. For example, if a JComboBox named historyChoice has been filled with a list of historical events, such as "Declaration of Independence," "Pearl Harbor," and "Man walks on moon," you can code the following to retrieve the user's choice:

```
int positionOfSelection = historyChoice.getSelectedIndex();
```

The variable positionOfSelection now holds the position of the selected item, and you can use the variable to access an array of dates so you can display the date that corresponds to the selected historical event. For example, if you declare the following, then dates[positionOfSelection] holds the year for the selected historical event:

```
int[] dates = {1776, 1941, 1969};
```

A JComboBox does not have to hold items declared as Strings; it can hold an array of Objects and display the results of the toString() method used with those objects. In other words, instead of using parallel arrays to store historical events and dates, you could design a HistoricalEvent class that encapsulates Strings for the event and ints for the date.

In addition to JComboBoxes for which users click items presented in a list, you can create JComboBoxes into which users type text. To do this, you use the setEditable() method. A drawback to using an editable JComboBox is that the text a user types must exactly match an item in the list box. If the user misspells the selection or uses the wrong case, no valid value is returned from the getSelectedIndex() method. You can use an if statement to test the value returned from getSelectedIndex(); if it is negative, the selection did not match any items in the JComboBox, and you can issue an appropriate error message.

TWO TRUTHS & A LIE

Using the JCheckBox, ButtonGroup, and JComboBox Classes

1. A JCheckBox consists of a label positioned beside a square; you can click the square to display or remove a check mark.
2. When you create a ButtonGroup, you can group several components, such as JCheckBoxes, so a user can select multiple options simultaneously.
3. When a user clicks a JComboBox, a list of alternative items drops down; if the user selects one, it replaces the box's displayed item.

The false statement is #2. When you create a ButtonGroup, you can group several components, such as JCheckBoxes, so a user can select only one at a time.

You Do It

Including JCheckBoxes in an Application

Next, you create an interactive program for a resort. The base price for a room is $200, and a guest can choose from several options. Reserving a room for a weekend night adds $100 to the price, including breakfast adds $20, and including a round of golf adds $75. A guest can select none, some, or all of these premium additions. Each time the user changes the option package, the price is recalculated.

1. Open a new file in your text editor, then type the following first few lines of a Swing application that demonstrates the use of a JCheckBox. Note that the JResortCalculator class implements the ItemListener interface:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JResortCalculator extends
   JFrame implements ItemListener
{
```

2. Declare the named constants that hold the base price for a resort room and the premium amounts for a weekend stay, including breakfast and a round of golf. Also include a variable that holds the total price for the stay, and initialize it to the value of the base price. Later, depending on the user's selections, premium fees might be added to totalPrice, making it more than BASE_PRICE.

```
final int BASE_PRICE = 200;
final int WEEKEND_PREMIUM = 100;
final int BREAKFAST_PREMIUM = 20;
final int GOLF_PREMIUM = 75;
int totalPrice = BASE_PRICE;
```

3. Declare three JCheckBox objects. Each is labeled with a String that contains a description of the option and the cost of the option. Each JCheckBox starts unchecked or deselected.

```
JCheckBox weekendBox = new JCheckBox
   ("Weekend premium $" + WEEKEND_PREMIUM, false);
JCheckBox breakfastBox = new
   JCheckBox("Breakfast $" + BREAKFAST_PREMIUM, false);
JCheckBox golfBox = new JCheckBox
   ("Golf $" + GOLF_PREMIUM, false);
```

(continues)

(continued)

4. Include `JLabel`s to hold user instructions and information and a `JTextField` in which to display the total price:

```
JLabel resortLabel = new JLabel
   ("Resort Price Calculator");
JLabel ePrice = new JLabel("The price for your stay is");
JTextField totPrice = new JTextField(4);
JLabel optionExplainLabel = new JLabel
   ("Base price for a room is $"
   + BASE_PRICE + ".");
JLabel optionExplainLabel2 = new JLabel
   ("Check the options you want.");
```

5. Begin the `JResortCalculator` class constructor. Include instructions to set the title by passing it to the `JFrame` parent class constructor, to set the default close operation, and to set the layout manager. Then add all the necessary components to the `JFrame`.

```
public JResortCalculator()
{
      super("Resort Price Estimator");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new FlowLayout());
      add(resortLabel);
      add(optionExplainLabel);
      add(optionExplainLabel2);
      add(weekendBox);
      add(breakfastBox);
      add(golfBox);
      add(priceLabel);
      add(totPrice);
```

6. Continue the constructor by setting the text of the `totPrice JTextField` to display a dollar sign and the `totalPrice` value. Register the class as a listener for events generated by each of the three `JCheckBox`es. Finally, add a closing curly brace for the constructor.

```
   totPrice.setText("$" + totalPrice);
   weekendBox.addItemListener(this);
   breakfastBox.addItemListener(this);
   golfBox.addItemListener(this);
}
```

(continues)

(continued)

7. Begin the `itemStateChanged()` method that executes when the user selects or deselects a `JCheckBox`. Use the appropriate methods to determine which `JCheckBox` is the source of the current `ItemEvent` and whether the event was generated by selecting a `JCheckBox` or by deselecting one.

```
public void itemStateChanged(ItemEvent event)
{
   Object source = event.getSource();
   int select = event.getStateChange();
```

8. Write a nested `if` statement that tests whether the source is equivalent to the `weekendBox`, `breakfastBox`, or, by default, the `golfBox`. In each case, depending on whether the item was selected or deselected, add or subtract the corresponding premium fee from the `totalPrice`. Display the total price in the `JTextField`, and add a closing curly brace for the method.

```
   if(source == weekendBox)
      if(select == ItemEvent.SELECTED)
         totalPrice += WEEKEND_PREMIUM;
      else
         totalPrice -= WEEKEND_PREMIUM;
    else if(source == breakfastBox)
    {
      if(select == ItemEvent.SELECTED)
         totalPrice += BREAKFAST_PREMIUM;
      else
         totalPrice -= BREAKFAST_PREMIUM;
    }
    else     // if(source == golfBox) by default
      if(select == ItemEvent.SELECTED)
         totalPrice += GOLF_PREMIUM;
      else
         totalPrice -= GOLF_PREMIUM;
      totPrice.setText("$" + totalPrice);
   }
```

9. Add a `main()` method that creates an instance of the `JFrame` and sets its size and visibility. Then add a closing curly brace for the class.

```
   public static void main(String[] args)
   {
      JResortCalculator aFrame =
         new JResortCalculator();
      final int WIDTH = 300;
      final int HEIGHT = 200;
      aFrame.setSize(WIDTH, HEIGHT);
      aFrame.setVisible(true);
   }
}
```

(continues)

(continued)

10. Save the file as **JResortCalculator.java**. Compile and execute the application. The output appears in Figure 14-41 with the base price initially set to $200.

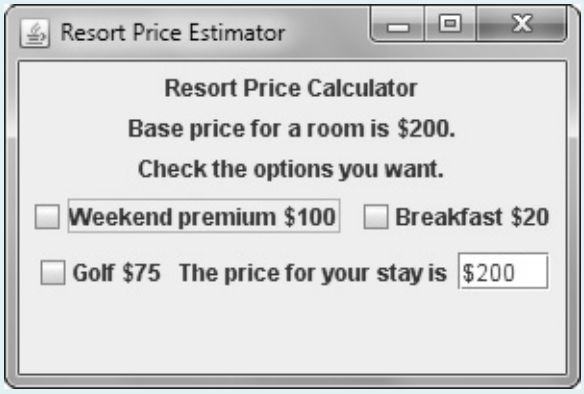

Figure 14-41 Initial output of the JResortCalculator application

11. Select the **Weekend premium** JCheckBox, and note the change in the total price of the event. Experiment with selecting and deselecting options to ensure that the price changes correctly. For example, Figure 14-42 shows the application with the weekend and golf options selected, adding a total of $175 to the $200 base price. After testing all the option combinations, close the application.

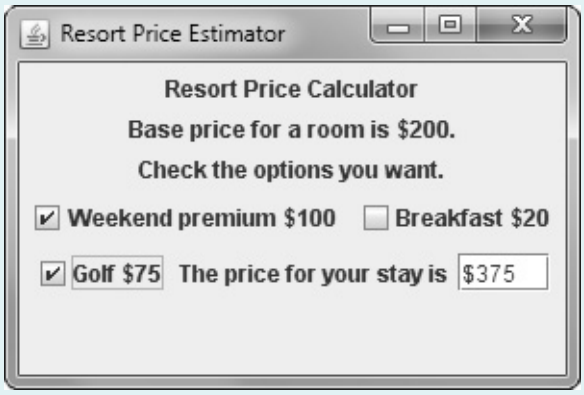

Figure 14-42 Output of the JResortCalculator application after the user has made selections

Don't Do It

- Don't forget the *x* in `javax` when you import `Swing` components into an application.
- Don't forget to use a `JFrame`'s `setVisible()` method if you want the `JFrame` to be visible.
- Don't forget to use `setLayout()` when you add multiple components to a `JFrame`.
- Don't forget to call `validate()` and `repaint()` after you add or remove a component from a container that has been made visible.
- Don't forget that the `ButtonGroup` class does not begin with a *J*.

Key Terms

UI components are user interface components, such as buttons and text fields, with which the user can interact.

`Swing` components are UI elements such as dialog boxes and buttons; you can usually recognize their names because they begin with *J*.

Java Foundation Classes, or **JFC**, include `Swing` component classes and selected classes from the `java.awt` package.

Lightweight components are written completely in Java and do not have to rely on the code written to run the local operating system.

Heavyweight components require interaction with the local operating system.

A **container** is a type of component that holds other components so you can treat a group of them as a single entity. Often, a container takes the form of a window that you can drag, resize, minimize, restore, and close.

A **`JFrame`** is a container with a title bar and border.

Pixels are the picture elements, or tiny dots of light, that make up the image on your computer monitor.

Window decorations are the icons and buttons that are part of a window or frame.

A **look and feel** is the default appearance and behavior of any user interface.

`JLabel` is a built-in Java `Swing` class that holds text you can display.

The **`add()` method** adds components to a container.

The **`remove()` method** removes components from a container.

The **`setText()` method** allows you to change the text in a `Component` that has already been created.

The **`getText()` method** allows you to retrieve the `String` of text in a `Component`.

The **`Font` class** holds typeface and size information.

The **`setFont()` method** changes a `JLabel`'s font.

The **typeface argument** to the `Font` constructor is a `String` representing a font. Common fonts have names such as Arial, Century, Monospaced, and Times New Roman.

The **style argument** to the `Font` constructor applies an attribute to displayed text and is one of three values: `Font.PLAIN`, `Font.BOLD`, or `Font.ITALIC`.

The **point size argument** to the `Font` constructor is an integer that represents about 1/72 of an inch.

A **layout manager** is a class that controls component positioning.

A **`BorderLayout`** is a layout manager that divides a container into regions.

The **flow layout manager** places components in a row, and when a row is filled, components automatically spill into the next row. By default, the components in each row are centered.

A **`JTextField`** is a component into which a user can type a single line of text data.

Editable describes a component that can accept keystrokes.

The **`setEditable()` method** changes the editable status of a `JTextField`.

A **`JButton`** is a `Component` the user can click with a mouse to make a selection.

Tool tips are popup windows that can help a user understand the purpose of components in an application; a tool tip appears when a user hovers the mouse pointer over the component.

The **`setToolTipText()` method** defines the text to be displayed in a tool tip.

An **event** occurs when a user takes action on a component.

In an **event-driven program**, the user might initiate any number of events in any order.

The **source** of an event is the component on which an event is generated.

A **listener** is an object that is interested in an event.

You tell a class to expect `ActionEvents` with the **`addActionListener()` method**.

The **`actionPerformed(ActionEvent e)` method** specification defines the actions that occur in response to an event.

The **`setEnabled()` method** makes a component available or dimmed and unavailable.

An **event handler** is a method that executes because it is called automatically when an appropriate event occurs.

A **JCheckBox** consists of a label positioned beside a square; you can click the square to display or remove a check mark. Usually, you use a `JCheckBox` to allow the user to turn an option on or off.

A **ButtonGroup** groups several components, such as `JCheckBoxes`, so a user can select only one at a time.

A **JComboBox** is a component that combines two features: a display area showing a default option and a list box containing additional options. The display area contains either a button that a user can click or an editable field into which the user can type.

Generic programming is a feature of languages that allows methods to be used safely with multiple data types.

Chapter Summary

- `Swing` components are UI elements such as dialog boxes and buttons. Each `Swing` component is a descendant of a `JComponent`, which in turn inherits from the `java.awt.Container` class. `Swing` components usually are placed in a container—a type of component that holds other components. Containers are defined in the `Container` class. Often, a container takes the form of a window that you can drag, resize, minimize, restore, and close.
- A `JFrame` holds and displays other objects. Useful methods include `setSize()`, `setTitle()`, `setVisible()`, `setBounds()`, and `setDefaultCloseOperation()`. `JFrames` include a title bar at the top containing text information, and Minimize, Maximize or Restore, and Close buttons in the frame's upper-right corner. When a user closes a `JFrame` by clicking the Close button in the upper-right corner, the default behavior is for the `JFrame` to become hidden and for the application to keep running.
- `JLabel` is a built-in Java `Swing` class that holds text you can display. You then can add a `JLabel` to a `JFrame` using the `add()` method. The `setFont()` method changes the font typeface, style, and point size.
- To place multiple components at specified positions in a container so they do not hide each other, you must use a layout manager—a class that controls component positioning. The normal (default) behavior of a `JFrame` is to use a layout format named `BorderLayout`. When you use `FlowLayout`, components do not lie on top of each other. Instead, components are placed in a row, and when a row is filled, components automatically spill into the next row.
- The advantage of creating a child class of `JFrame` is that you can set the `JFrame`'s properties within your object's constructor; then, when you create your `JFrame` child object, it is automatically endowed with the features you have specified, such as title, size, and default close operation.

- A `JTextField` is a component into which a user can type a single line of text data. A `JButton` is a `Component` the user can click with a mouse to make a selection. Tool tips are popup windows that can help a user understand the purpose of components in an application; the tool tip appears when a user hovers the mouse pointer over the component.
- Within an event-driven program, a component on which an event is generated is the source of the event. An object that is interested in an event is a listener. You prepare your class to accept button-press events by importing the `java.awt.event` package into your program and adding the phrase `implements ActionListener` to the class header. You tell your class to expect `ActionEvents` with the `addActionListener()` method. The `ActionListener` interface contains the `actionPerformed(ActionEvent e)` method specification. You implement this method with the actions that should occur in response to the event. Within the `actionPerformed()` method, you can use the `getSource()` method of the object sent to determine which component generated the event.
- A class can implement as many event listeners as it needs. Examples of event listeners are `ActionListener`, `ItemListener`, `KeyListener`, and `MouseListener`. Any object can be notified of an event as long as it implements the appropriate interface and is registered as an event listener on the appropriate event source. To add a listener method to a source, you must use the appropriate designated `add()` method. Specific methods react to specific event types; they include `actionPerformed()` and `itemStateChanged()`.
- A `JCheckBox` consists of a label positioned beside a square; you can click the square to display or remove a check mark. Usually, you use a `JCheckBox` to allow the user to turn an option on or off. A `ButtonGroup` groups components so a user can select only one at a time. After a `JCheckBox` in a `ButtonGroup` has been selected, one in the group will always be selected. A `JComboBox` is a component that combines two features: a display area showing a default option and a list box containing additional options. You can treat the list of items in a `JComboBox` object as an array and use the `getSelectedIndex()` method to determine the list position of the currently selected item.

Review Questions

1. A `JFrame` is a descendant of each of the following classes except the _____ class.

   a. `Component`
   b. `Jar`
   c. `Container`
   d. `Window`

2. A programmer might prefer using a `JFrame` instead of a `Window` because, unlike a window, a `JFrame` _____.

   a. can hold other objects
   b. can be made visible
   c. can have descendants
   d. has a title bar and border

3. The statement `JFrame myFrame = new JFrame();` creates a `JFrame` that is _____.

   a. invisible and has no title
   b. invisible and has a title
   c. visible and has no title
   d. visible and has a title

4. To create a `JFrame` named `aFrame` that is 300 pixels wide by 200 pixels tall, you can _____.

   a. use the declaration `JFrame aFrame = new JFrame(300, 200);`
   b. declare a `JFrame` named `aFrame` and then code `aFrame.setSize(300, 200);`
   c. declare a `JFrame` named `aFrame` and then code `aFrame.setBounds(300, 200);`
   d. use any of the above

5. When a user closes a `JFrame`, the default behavior is for _____.

   a. the `JFrame` to close and the application to keep running
   b. the `JFrame` to become hidden and the application to keep running
   c. the `JFrame` to close and the application to exit
   d. nothing to happen

6. An advantage of extending the `JFrame` class is _____.

   a. you can set the child class properties within the class constructor
   b. there is no other way to cause an application to close when the user clicks a `JFrame`'s Close button
   c. there is no other way to make a `JFrame` visible
   d. all of the above

7. Suppose that you create an application in which you instantiate a `JFrame` named `frame1` and a `JLabel` named `label1`. Which of the following statements within the application adds `label1` to `frame1`?

   a. `label1.add(frame1);`
   b. `frame1.add(label1);`
   c. `this.add(label1);`
   d. two of the above

8. The arguments required by the `Font` constructor include all of the following except _____.

   a. typeface
   b. style
   c. mode
   d. point size

9. A class that controls component positioning in a `JFrame` is a _____.

   a. container
   b. layout manager
   c. formatter
   d. design supervisor

10. Which of the following is not true of a `JTextField`?

    a. A user can type text data into it.
    b. Its data can be set in the program instead of by the user.
    c. A program can set its attributes so that a user cannot type in it.
    d. It is a type of `Container`.

11. _____ are popup windows that can help a user understand the purpose of components in an application and that appear when a user hovers the mouse pointer over the component.

    a. Navigation notes
    b. Tool tips
    c. Help icons
    d. Graphic suggestions

12. Within an event-driven program, a component on which an event is generated is the _____.

    a. performer
    b. listener
    c. source
    d. handler

13. A class that will respond to button-press events must use which phrase in its header?

    a. `import java.event`
    b. `extends Action`
    c. `extends JFrame`
    d. `implements ActionListener`

14. A `JFrame` contains a `JButton` named `button1` that should execute an `actionPerformed()` method when clicked. Which statement is needed in the `JFrame` class?

    a. `addActionListener(this);`
    b. `addActionListener(button1);`
    c. `button1.addActionListener(this);`
    d. `this.addActionListener(button1);`

15. When you use the `getSource()` method with an `ActionEvent` object, the result is _____.

    a. an `Object`
    b. an `ActionEvent`
    c. a `Component`
    d. a `TextField`

16. A class can implement _____.

    a. one listener
    b. two listeners
    c. as many listeners as it needs
    d. any number of listeners as long as they are not conflicting listeners

17. When you write a method that reacts to `JCheckBox` changes, you name the method _____.

    a. `itemStateChanged()`
    b. `actionPerformed()`
    c. `checkBoxChanged()`
    d. any legal identifier you choose

18. If a class contains two components that might each generate a specific event type, you can determine which component caused the event by using the _____ method.

    a. `addActionListener()`
    b. `getSource()`
    c. `whichOne()`
    d. `identifyOrigin()`

19. To group several components such as `JCheckBox`es so that a user can select only one at a time, you create a _____.

    a. `JCheckBoxGroup`
    b. `CheckBoxGroup`
    c. `JButtonGroup`
    d. `ButtonGroup`

20. Suppose that you have declared a `ButtonGroup` named `threeOptions` and added three `JCheckBox`es named `box1`, `box2`, and `box3` to it. If you code `threeOptions.setSelected(box1);`, then `threeOptions.setSelected(box2);`, and then `threeOptions.setSelected(box3);`, the selected box is _____.

    a. `box1`
    b. `box2`
    c. `box3`
    d. all of the above

Exercises

Programming Exercises

1. Write an application that displays a `JFrame` containing the first few lines of your favorite song. Save the file as **JLyrics.java**.

2. a. Write an application that instantiates a `JFrame` that contains a `JButton`. Disable the `JButton` after the user clicks it. Save the file as **JFrameDisableButton.java**.

   b. Modify the `JFrameDisableButton` program so that the `JButton` is not disabled until the user has clicked at least eight times. At that point, display a `JLabel` that indicates "That's enough!". Save the file as **JFrameDisableButton2.java**.

3. Create an application with a `JFrame` and five labels that contain the names of five friends. Every time the user clicks a `JButton`, remove one of the labels and add a different one. Save the file as **JDisappearingFriends.java**.

4. Write an application for Lambert's Vacation Rentals. Use separate `ButtonGroups` to allow a client to select one of three locations, the number of bedrooms, and whether meals are included in the rental. Assume that the locations are parkside for $600 per week, poolside for $750 per week, or lakeside for $825 per week. Assume that the rentals have one, two, or three bedrooms and that each bedroom over one adds $75 to the base price. Assume that if meals are added, the price is $200 more per rental. Save the file as **JVacationRental**.

5. a. Write an application that allows a user to select one of at least 12 songs from a combo box. Display the purchase price, which is different for each song. Save the file as **JTunes.java**.

   b. Change the `JTunes` application to include an editable combo box. Allow the user to type the name of a song to purchase. Display an appropriate error message if the desired song is not available. Save the file as **JTunes2.java**.

6. Design an application for a pizzeria. The user makes pizza order choices from list boxes, and the application displays the price. The user can choose a pizza size of small ($7), medium ($9), large ($11), or extra large ($14), and one of any number of toppings. There is no additional charge for cheese, but any other topping adds $1 to the base price. Offer at least five different topping choices. Save the file as **JPizza.java**.

7. Write an application that allows a user to select a city from a list box that contains at least seven options. Display the population of the city in a text field after the user makes a selection. Save the file as **JPopulation.java**.

8. Write an application that allows the user to choose insurance options in `JCheckBoxes`. Use a `ButtonGroup` to allow the user to select only one of two insurance types—HMO (health maintenance organization) or PPO (preferred provider organization). Use regular (single) `JCheckBoxes` for dental insurance and vision insurance options; the user can select one option, both options, or neither option. As the user selects each option, display its name and price in a text field; the HMO costs $200 per month, the PPO costs $600 per month, the dental coverage adds $75 per month, and the vision care adds $20 per month. When a user deselects an item, make the text field blank. Save the file as **JInsurance.java**.

9. a. Search the Java Web site for information on how to use a `JTextArea`, its constructors, and its `setText()` and `append()` methods. Write an application that allows the user to select options for a dormitory room. Use `JCheckBoxes` for options such as private room, Internet connection, cable TV connection, microwave, refrigerator, and so on. When the application starts, use a text area to display a message listing the options that are not yet selected. As the user selects and deselects options, add appropriate messages to the common text area so it accumulates a running list that reflects the user's choices. Save the file as **JDorm.java**.

b. Modify the `JDorm` application so that instead of a running list of the user's choices, the application displays only the current choices. Save the file as **JDorm2.java**.

10. Create an application for Paula's Portraits, a photography studio. Paula's base price is $40 for a photo session with one person. The in-studio fee is $75 for a session with two or more subjects, and $95 for a session with a pet. A $90 fee is added to take photos on location instead of in the studio. The application allows users to compute the price of a photography session. Include a set of mutually exclusive check boxes to select the portrait subject and another set of mutually exclusive check boxes for the session location. Include labels as appropriate to explain the application's functionality. Save the file as **JPhotoFrame.java**.

Debugging Exercises

1. Each of the following files in the Chapter14 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugFourteen1.java will become FixDebugFourteen1.java.

a. DebugFourteen1.java
b. DebugFourteen2.java
c. DebugFourteen3.java
d. DebugFourteen4.java

Game Zone

1. a. Create a quiz game that displays, in turn, five questions about any topic of your choice. All five questions should have the same three possible multiple-choice answers. For example, you might ask trivia questions about U.S. states for which the correct response is either California, Florida, or New York. After each question is displayed, allow the user to choose one, two, or all three answers by selecting `JCheckBoxes`. In other words, if the user is sure of an answer, he will select just one box, but if he is uncertain, he might select two or three boxes. When the user is ready to submit the answer(s), he clicks a button. If the user's answer to the question is correct and he has selected just one box, award 5 points. If the user is correct but has selected two boxes, award 2 points. If the user has selected all three boxes, award 1 point. If the user has selected fewer than three boxes but is incorrect, the user receives 0 points. A total of 25 points is possible. If the user has accumulated more than 21 points at the end of the quiz, display the message "Fantastic!". If the user has accumulated more than 15 points, display the message "Very good", and if the user has accumulated fewer points, display "OK". Save the file as **HedgeYourBet.java**.

b. Modify the `HedgeYourBet` game so that it stores the player's score from the last game in a file and displays the previous score at the start of each new game. (The first time you play the game, the previous score should be 0.) Save the game as **HedgeYourBetUsingFile.java**.

2. In Chapter 5, you created a lottery game application. Create a similar game using check boxes. For this game, generate six random numbers, each between 0 and 30, inclusive. Allow the user to choose six check boxes to play the game. (Do not allow the user to choose more than six boxes.) After the player has chosen six numbers, display the randomly selected numbers, the player's numbers, and the amount of money the user has won, as follows:

Matching Numbers	Award ($)
Three matches	100
Four matches	10,000
Five matches	50,000
Six matches	1,000,000
Zero, one, or two matches	0

Save the file as **JLottery2.java**.

3. a. Create a game called Last Man Standing in which the objective is to select the last remaining `JCheckBox`. The game contains 10 `JCheckBoxes`. The player can choose one, two, or three boxes, and then click a `JButton` to indicate the turn is complete. The computer then randomly selects one, two, or three `JCheckBox` objects. When the last `JCheckBox` is selected, display a message indicating the winner. Save the game as **LastManStanding.java**.

   b. In the current version of the Last Man Standing game, the computer might seem to make strategic mistakes because of its random selections. For example, when only two `JCheckBox` objects are left, the computer might randomly choose to check only one, allowing the player to check the last one and win. Modify the game to make it as smart as possible, using a random value for the number of the computer's selections only when there is no superior alternative. Save the improved game as **SmarterLastManStanding.java**.

Case Problems

1. In previous chapters, you have created a number of programs for Carly's Catering. Now, create an interactive program that allows the user to enter the number of guests for an event into a text field; if the value entered is not numeric, set the event price to 0. Also allow the user to choose one entree from a group of at least four choices, up to two side dishes from a group of at least four choices, and one

dessert from a group of at least three choices. Display the cost of the event as $35 per person; as the user continues to select and deselect menu items, display a list of the current items chosen. If a user attempts to choose more than two side dishes, remove all the current side dish selections so that the user can start over. Save the program as **JCarlysCatering.java**.

2. In previous chapters, you have created a number of programs for Sammy's Seashore Rentals. Now, create an interactive GUI program that allows the user to enter a rental time in hours into a text field; if the value entered is not numeric, set the rental price to 0. Also allow the user to choose one equipment type to rent from a group of seven choices. The rental fee is $40 per hour for a jet ski or pontoon boat; $20 per hour for a rowboat, canoe, or kayak; and $7 per hour for a beach chair or umbrella. Let the user add an equipment lesson for an extra $5. Display a message that indicates all the details for the rental, including the total price. Save the program as **JSammysSeashore.java**.