CHAPTER

9

APPLETS

In this chapter, you will:

- Write an HTML document to host an applet
- Understand simple applets
- Use Labels with simple AWT applets
- Write a simple Swing applet and use a JLabel
- Add JTextField and JButton Components to Swing applets
- Learn about event-driven programming
- Add output to a Swing applet
- Understand the Swing applet life cycle
- Create a more-sophisticated interactive Swing applet
- Use the setLocation() and setEnabled() methods

It seems like I've learned a lot," you tell Lynn Greenbrier during a coffee break at Event Handlers Incorporated. "I can use variables, make decisions, write loops, and use arrays."

"You've come a long way," Lynn agrees.

"But at the same time," you continue, "I feel like I know nothing! When I visit the simplest Web site, it looks far more sophisticated than my most advanced application. There is color and movement. There are buttons to click and boxes into which I can type responses to questions. Nothing I've done even approaches that."

"But you have a good foundation in Java programming," Lynn says. "Now you can put all that knowledge to work. By adding a few new objects to your repertoire, and by learning a little about applets, you can comfortably enter the world of interactive Web programming."

Previewing the AWT and Swing Greet Applets

The Chap9Greet and Chap9JGreet classes preview two basic applets. Chap9Greet is an applet created using the Abstract Windows Toolkit (AWT) and the Applet class. Chap9JGreet is also an applet created using the AWT, but is different because Chap9JGreet uses the JApplet class.

You can now use a completed version of each applet, which you will find saved in the Chapter.09 folder on your Student Disk. Also saved in the Chapter.09 folder are copies of Chap9Greet and Chap9JGreet.

To run the Chap9Greet and Chap9JGreet applets:

1. At the command prompt for the Chapter.09 folder on your Student Disk, type **`appletviewer TestChap9Greet.html`**, and then press **[Enter]**. It might take a few minutes for the Applet Viewer window to open. See Figure 9-1.

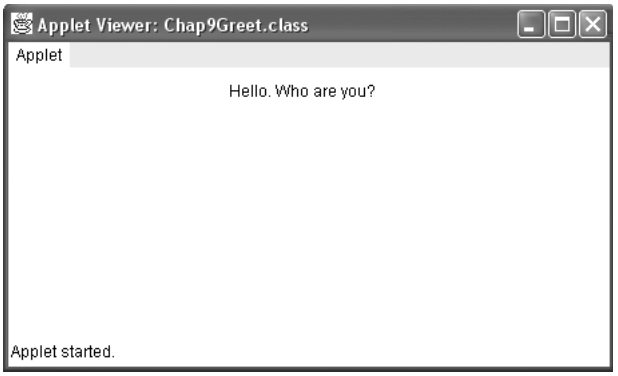

Figure 9-1 Chap9Greet applet

2. Click the **Close** button in the upper-right corner of the Applet Viewer window to close the Applet Viewer.
3. At the command prompt for the Chapter.09 folder on your Student Disk, type **`appletviewer TestChap9JGreet.html`**, and then press **[Enter]**. View the applet shown in Figure 9-2.
4. Close the Applet Viewer.

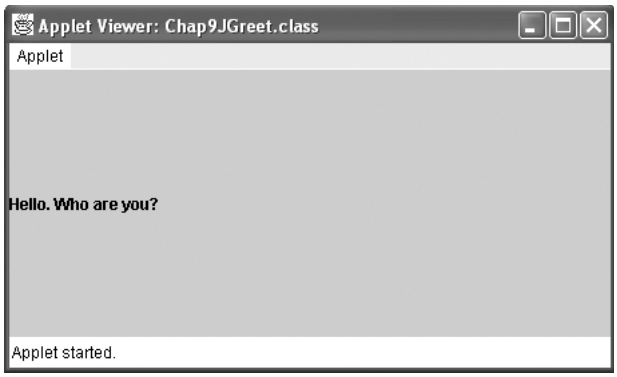

Figure 9-2 Chap9JGreet applet

Writing an HTML Document to Host an Applet

You have written many Java applications. When you write a Java application, you do the following:

- Write the application in the Java programming language, and then save it with a .java file extension.
- Compile the application into bytecode using the `javac` command. The bytecode is stored in a file with a .class file extension.
- Use the `java` command to interpret and execute the .class file.

As you know, applications are stand-alone programs. In contrast, **applets** are programs that are called from within another application. You run applets within a page on the Internet, an intranet, or a local computer from within another program called **Applet Viewer**, which comes with the Java Developer's Kit. To view an applet, it must be called from within another document written in HTML. **HTML**, **Hypertext Markup Language**, is a simple language used to create Web pages for the Internet. HTML contains many commands that allow you to format text on a Web page, import graphic images, and link your page to other Web pages. When you create an applet, you do the following:

- Write the applet in the Java programming language, and save it with a .java file extension, just as when you write a Java application.
- Compile the applet into bytecode using the `javac` command, just as when you write a Java application.
- Write an HTML document that includes a statement to call your compiled Java class.

- Load the HTML document into a Web browser (such as Netscape Navigator or Microsoft Internet Explorer), or run the Applet Viewer program, which, in turn, uses the HTML document.

Java, in general, and applets, in particular, are popular topics among programmers, mostly because users can execute applets using a Web browser on the Internet. A **Web browser** is a program that allows you to display HTML documents on your computer screen. Web documents often contain Java applets.

Fortunately, to run a Java applet, you don't need to learn the entire HTML language; you need to learn only two pairs of HTML commands, called **tags**. The tag that begins every HTML document is **<HTML>**. Like all tags, this tag is surrounded by angle brackets. `HTML` is an HTML keyword which specifies that an HTML document follows the keyword. The tag that ends every HTML document is **</HTML>**. Placing a backslash before any tag indicates the tag is the ending half of a pair of tags. The following is the simplest HTML document you can write:

```
<HTML>
</HTML>
```

Unlike the Java programming language, HTML is not case sensitive so you can use `<html>` in place of `<HTML>`. However, this book uses the all uppercase convention when typing HTML code. With the growing importance of XML and XHTML, many programmers recommend putting all tags in lowercase since XML and XHTML are case sensitive.

The simple HTML document begins and ends and does nothing in the process; you can create an analogous situation in a Java method by typing an opening curly brace and following it immediately with the closing curly brace. HTML documents generally contain more statements. For example, to run an applet from within an HTML document, you add an **<APPLET>** and **</APPLET>** tag pair. Usually, you place three attributes within the **<APPLET>** tag: `CODE`, `WIDTH`, and `HEIGHT`. **Attributes**, sometimes referred to as arguments, promote activity—with them the HTML tag can do something in a certain way. Note the following example:

```
<APPLET CODE = "AClass.class" WIDTH = 300 HEIGHT = 200
</APPLET>
```

The following are three `APPLET` tag attributes and a description of their corresponding arguments:

- `CODE =` is followed by the name of the compiled applet you are calling
- `WIDTH =` is followed by the width of the applet on the screen
- `HEIGHT =` is followed by the height of the applet on the screen

The name of the applet you call must be a compiled Java applet (with a .class file extension). The width and height of an applet are measured in pixels. **Pixels** are the picture

elements, or tiny dots of light that make up the image on your video monitor. For monitors that display 800 pixels horizontally and 600 pixels vertically, a statement such as `WIDTH = 400 HEIGHT = 300` will create an applet that occupies approximately one-fourth of most screens (half the height and half the width).

SVGA monitors commonly display 800 x 600 pixels, 1024 x 768 pixels, or higher. If you want most users to see larger applets without scrolling, the maximum size of your applets should be 760 x 520 pixels. The standard 600 x 400 pixels was used to support the older 640 x 480 VGA monitors and is currently considered out-of-date. Keep in mind that the browser's menu bar and screen elements (such as the toolbar and the scrollbars) will take up some of the screen viewing area for an applet.

Next you will create a simple HTML document that you will use to display the applet that you create in the next section. You will name the applet Greet, and it will occupy a screen area of 450 × 200 pixels.

To create a simple HTML document:

1. Open a new file in your text editor.
2. Type the opening HTML tag, **`<HTML>`**.
3. On the next line, type the opening `APPLET` tag that contains the applet's name and dimensions: **`<APPLET CODE = "Greet.class" WIDTH = 450 HEIGHT = 200>`**.
4. On the next line, type the applet's closing tag: **`</APPLET>`**.
5. On the next line, type the closing HTML tag: **`</HTML>`**.
6. Save the file as **TestGreet.html** in the Chapter.09 folder on your Student Disk. Just as when you create a Java application, make sure that you save the file as text only using an .html extension. The .html file extension is required and makes the file easy to identify as an HTML file. If you are using Notepad or another text editor, you can enclose the filename in quotation marks to save the .html file extension, as in "A:\Chapter.09\TestGreet.html".

Understanding Simple Applets

To write an applet you must learn only a few additions and changes to writing a Java application. In addition to what you learned about creating applets in the beginning of this chapter, you must also do the following to write an applet:

- Include import statements to ensure that necessary classes are available.
- Learn to use some Windows components and applet methods.
- Learn to use the keyword `extends`.

In Chapter 3, you used an import statement to access classes such as java.util.Date and java.util.GregorianCalendar within your application. You imported these classes to avoid having to write common date-handling routines that already exist in Java. Similarly, Java's creators fashioned a variety of classes to handle common applet needs. Programmers use core classes of the Java platform that include the java.applet.Applet and javax.swing.JApplet. The structure for these core classes is shown in Figure 9-3.

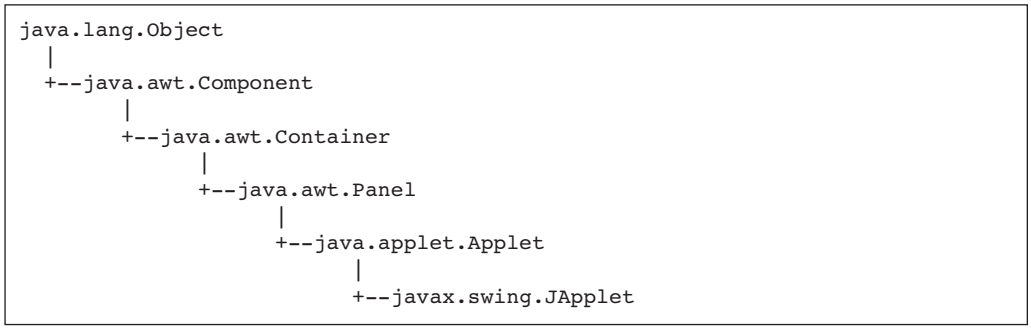

Figure 9-3 Structure of Applet and JApplet classes

The class structure in Figure 9-3 shows that the `Applet` class is used to create an AWT applet and the `JApplet` class is used to create a Swing applet. You can use methods from the Component and Container classes for both types. A **Component** is a class that defines any object that you want to display. In the pages that follow, you will display frames, panels, buttons, labels, and text fields in the applets you create. A **Container** is a class that is used to define a component that can contain other components.

Most AWT applets contain at least two import statements: `import java.applet.*;` and `import java.awt.*;`. The java.applet package contains a class named Applet—every applet you create is based on this class. The java.awt package is the Abstract Windows Toolkit, or AWT. (In Java 1.1, the java.awt package contained the primary classes you would use to create a Graphical User Interface, or GUI.)

Most Swing applets contain at least two import statements: `import javax.swing.*;` and `import java.awt.*;`. Many of the AWT classes have been superceded in Java 2 by Swing classes. The classes in the javax.swing package define GUI elements, referred to as **Swing components**, and provide much-improved alternatives to components defined by classes in java.awt. The Swing classes are part of a more general set of GUI programming capabilities that are collectively referred to as the **Java Foundation Classes**, or **JFC**. JFC includes Swing component classes and selected classes from the java.awt package.

When you create an application, you follow any needed import statements with a class header such as `public class AClass`. AWT applets and Swing applets begin the same way as Java applications, but they must also include the words `extends Applet`

and `extends JApplet`. The keyword `extends` indicates that your applet will build on, or inherit, the traits of the Applet or JApplet classes.

Both the Applet and JApplet classes provide a general outline used by any Web browser when it runs an applet. In an application, the main() method calls other methods that you write. With an applet, the browser calls many methods automatically. The following four methods are included in every applet:

- `public void init()`
- `public void start()`
- `public void stop()`
- `public void destroy()`

If you fail to write one or more of these methods, Java creates them for you. The methods Java creates have opening and closing curly braces only—in other words, they are empty. To create a Java program that does anything useful, you must code at least one of these methods.

The init() method is the first method called in any applet. You use it to perform initialization tasks, such as setting variables to initial values or placing applet components on the screen. You must code the init() method's header as `public void init()`.

In the text which follows, you will need to distinguish between applets created by the Applet and JApplet classes. The term Applet is used for the AWT applet; JApplet is the term used for the Swing applet. When the term applet is used alone, it refers to material that can apply to both applet types.

Using Labels with Simple AWT Applets

The java.awt package contains commonly used Windows components such as Labels, Menus, and Buttons. You import java.awt so you don't have to "reinvent the wheel" by creating these components yourself. AWT Applets are not required to contain Windows components, but they almost always do.

One of the simplest Window components is a Label. **Label** is a built-in class that holds text that you can display within an applet. The Label class also contains fields that indicate appearance information, such as font and alignment. As with other objects, you can declare a Label without allocating memory, as in `Label greeting;`, or you can call the Label constructor without any arguments, as in `Label greeting = new Label();`. You can assign some text to the Label with the **setText() method**, as in `greeting.setText("Hi there");`. Alternately, you can call the Label constructor and pass it a String argument so the Label is initialized upon construction, as in `Label greeting = new Label("Hello. Who are you?");`.

You use the **add() method** to add a component to an applet window. For example, if a Label is defined as `Label greeting = new Label("Hello. Who are you?");`, then you can place a greeting within an applet using the command `add(greeting);`.

The object of the add() method is the applet itself, so when you add a component to a window, you could write `this.add();` in place of `add();`. You learned about the `this` reference in Chapter 4.

Figure 9-4 shows the program to create an applet that displays "Hello. Who are you?" on the screen.

```
import java.applet.*;
import java.awt.*;

public class Greet extends Applet
{
   Label greeting = new Label("Hello. Who are you?");
     public void init()
     {
        add(greeting);
     }
}
```

Figure 9-4 AWT Greet applet

Next you will create and compile the Greet applet.

To create and run the Greet applet:

1. Open a new text file in your text editor.
2. Enter the code shown in Figure 9-4.
3. Save the file as **Greet.java** in the Chapter.09 folder on your Student Disk.
4. Compile the program with the command **`javac Greet.java`**.
5. If necessary, correct any errors, and then compile again.

To run the Greet applet, you can use your Web browser or the `appletviewer` command. In the following steps, you will do both.

To run the applet using your Web browser:

1. Open any Web browser, such as Microsoft Internet Explorer or Netscape. You do not have to connect to the Internet; you will use the browser locally.

If you do not have a Web browser installed on your computer, skip to the end of Step 3.

2. Click **File** on the menu bar, click **Open** or **Open Page**, type **A:\Chapter.09\TestGreet.html**, which is the complete path for the HTML document that you created to access Greet.class, and then press **[Enter]**. The applet should appear on your screen, as shown in Figure 9-5. If you receive an error message, verify that the path and spelling of the HTML file are correct.

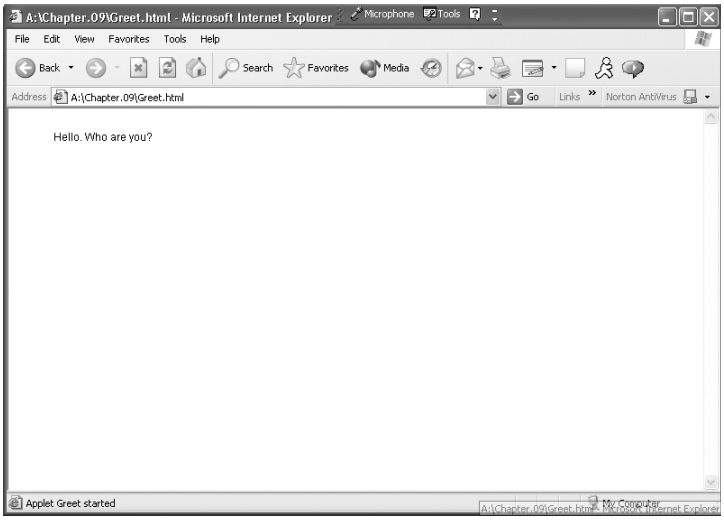

Figure 9-5 TestGreet.html displayed in Internet Explorer

3. Click the **Close button** in the upper right-hand corner of the browser's program window to close your Web browser.

You can also view your applet using the `appletviewer` command. In this book, you will test your applets using this command.

Some applets may not work correctly using your browser. Java was designed with a number of security features so that when an applet displays on the Internet, the applet cannot perform malicious tasks, such as deleting a file from your hard drive. If an applet does nothing to compromise security, then testing it using the Web browser or the `appletviewer` command achieves the same results. For now, you can get your applets to perform better by using the Applet Viewer window because the output will not depend on browser type or version.

To run the applet using the `appletviewer` command:

1. At the command line, type **`appletviewer TestGreet.html`**, and then press **[Enter]**. After a few moments, the Applet Viewer window opens and displays the applet, as shown in Figure 9-6.

2. Use the mouse pointer to drag any corner of the Applet Viewer window to resize it. Notice that if you widen the window by dragging its right border to the right, the window is redrawn on the screen and the Label is automatically repositioned to remain centered within the window. If you narrow the window by dragging its left border to the left, the Label eventually is partially obscured when the window becomes too narrow for the display.

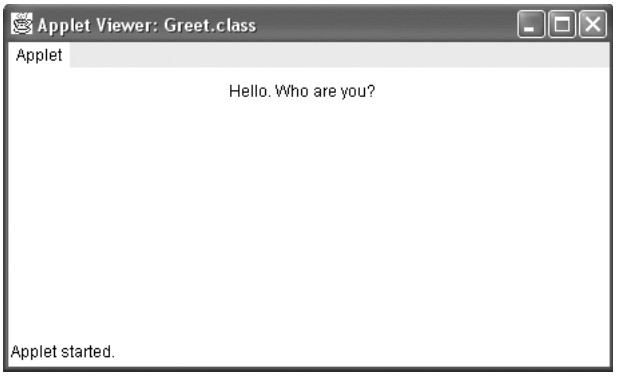

Figure 9-6 Output of TestGreet.html page in an Applet Viewer window

3. Close the Applet Viewer.

Writing a Simple Swing Applet and Using a JLabel

The Swing component classes offer more flexibility than classes defined in the java.awt package because they are implemented entirely in Java. Older java.awt components greatly relied on the native code of the operating system on which Java was then implemented. This chapter makes primary use of the GUI built on the Java Foundation classes. You will import java.awt and javax.swing classes so you don't have to create these components yourself.

The counterpart to the AWT Label is a JLabel. **JLabel** is a built-in class that holds text that you can display within an applet. The structure of the JLabel class is shown in Figure 9-7.

Available constructors for the JLabel class include:

- `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, inthorizontalAlignment)` 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.

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

Figure 9-7 Structure of the JLabel class

Using the JLabel class, you allocate memory, create objects, and assign text using class constructors and methods that are virtually identical to those found in the Label class. The primary difference is that more constructors and methods are available in the JLabel class.

A major difference between AWT and Swing classes is the method used to add a component to a Swing window. AWT components are added directly to the Applet; Swing components must use a content pane. GUI components, such as JLabels, are attached to a separate content pane so they can be displayed at execution time. This is true for a Swing application as well as a Swing applet.

Referring back to Figure 9-3, the **content pane** is an object of the Container class from the java.awt package. A Container object can be created using the **getContentPane() method**. To create a Container object named `con`, the syntax is `Container con = getContentPane();`. Then the statement `JLabel greeting = new JLabel();` adds the greeting object to the content pane with the statement `con.add(greeting);`.

Figure 9-8 shows the program to create a Swing applet that displays "Hello. Who are you?" on the screen. Next you will create a Swing applet.

To create and run the JGreet applet:

1. Open a new text file in your text editor.
2. Enter the code shown in Figure 9-8.
3. Save the file as **JGreet.java** in the Chapter.09 folder on your Student Disk.
4. Compile the program and, if necessary, correct any errors, and then compile again.

5. Open a new file in your text editor, and then type the HTML document that will run the JApplet as follows:

```
<HTML>
<APPLET CODE = "Greet.class" WIDTH = 450 HEIGHT = 200>
</APPLET>
</HTML>
```

   Save the file as **TestJGreet.html** in the Chapter.09 folder of your Student Disk.

6. At the command line, type **`appletviewer TestJGreet.html`**, and then press **[Enter]**. The applet appears on your screen, as shown in Figure 9-9.

```
import javax.swing.*;
import java.awt.*;
public class JGreet extends JApplet
{
   JLabel greeting = new JLabel("Hello. Who are you?");

   public void init()
   {
     Container con = getContentPane();
     con.add(greeting);
   }
}
```

Figure 9-8 Swing Greet applet

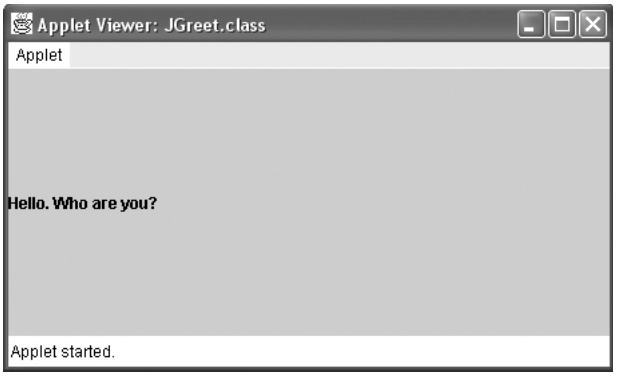

Figure 9-9 Output of TestJGreet.html page in Applet Viewer window

7. Close the Applet Viewer window.

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 either of your simple java applets. You might think that the string, "Hello. Who are you?" is pretty plain and lackluster. Fortunately, Java provides you with a **Font 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** is a String representing a font. Common fonts are Arial, Helvetica, Courier, and Times New Roman. The typeface is only a request; the system on which your applet runs might not have access to the requested font and substitute a default font. The **style** applies an attribute to displayed text and is one of three arguments, Font.PLAIN, Font.BOLD, or Font.ITALIC. The **point size** is an integer that represents ½ of an inch. Printed text is usually about 12 points; a headline might be 30 points.

To give a Label object a new font, you create the Font object, as in `Font headlineFont = new Font("Helvetica", Font.BOLD, 36);`, and then you use the setFont() method to assign the font to a Label with the statement `greeting.setFont(headlineFont);`.

The typeface name is a String, so you must enclose it in double quotation marks when you use it to declare the Font object.

Next you will change the font of the text in your JGreet applet.

To change the appearance of the greeting in the JGreet applet:

1. Open the **JGreet.java** file in your text editor and change the class name to **JGreet2**.
2. Position the insertion point at the end of the line that declares the greeting Label, and then press **[Enter]** to start a new line of text.
3. Declare a Font object named bigFont by typing the following:

   ```
   Font bigFont = new Font("TimesRoman", Font.ITALIC, 24);
   ```

4. Place the insertion point to the right of the opening curly brace of the init() method, and then press **[Enter]** to start a new line.
5. Set the greeting font to bigFont by typing `greeting.setFont(bigFont);`.
6. Save the file using the filename **JGreet2.java**.
7. At the command line, compile the program, and if necessary, correct any errors.
8. Run the applet, changing the TestJGreet.html document created earlier to **TestJGreet2.html**. Within the TestJGreet2.html document, change the class internally to `JGreet2.class` and then execute the `appletviewer TestJGreet2.html` command. Figure 9-10 shows the output.
9. Close the Applet Viewer window.

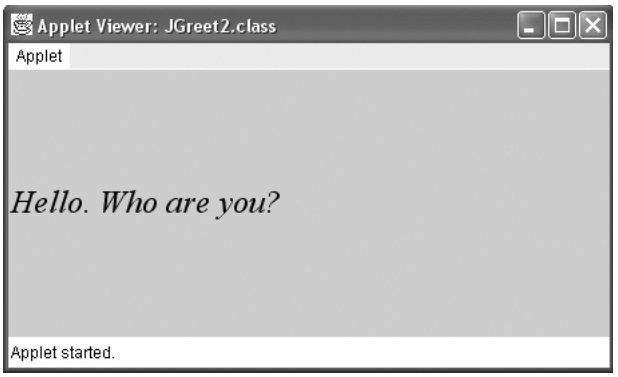

Figure 9-10 Output of the JGreet2.java program using bigFont

Adding JTextField and JButton Components to Swing Applets

In addition to including JLabels, Swing applets often contain other window features such as JTextFields and JButtons. 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.) The structure of the JTextField class is shown in Figure 9-11.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.text.JTextComponent
                          |
                          +--javax.swing.JTextField
```

Figure 9-11 Structure of the JTextField class

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 numColumns)` 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 for a user to answer the "Who are you?" question, you can code `JTextField answer = new TextField(10);` to provide a JTextField that is empty and displays approximately 10 characters. To add the JTextField named answer to an applet, you write `con.add(answer);`, where `con` is a Container object declared as `Container con = getContentPane();`.

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 Arial font can display 24 i characters, but only eight w characters.

Try to anticipate how many characters your users will 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 err on the high side when estimating the size of a user text field.

Several other methods are available for use with JTextFields. The **setText() method** allows you to change the text in a JTextField that has already been created, as in `answer.setText("Thank you");`. The **getText() method** allows you to retrieve the String of text in a JTextField, as in `String whatDidTheySay = answer.getText();`.

When a user encounters a JTextField you have placed within an applet, the user must position the mouse pointer in the JTextField and click to get an insertion point. When the user clicks within the JTextField, the JTextField has **keyboard focus**, which means that the next entries from the keyboard will be entered at that location. When you want the insertion point to appear automatically within the TextField without requiring the user to click in it first, you can use the **requestFocus() method**. For example, if you have added a JTextField named answer to an applet, then `answer.requestFocus()` causes the insertion point to appear within the JTextField, and the user can begin typing immediately without moving the mouse. In addition to saving the user some time and effort, requestFocus() is useful when you have several JTextFields and you want to direct the user's attention to a specific one. However, at any time, only one component within a window can have the keyboard focus.

When a JTextField has the capability of accepting keystrokes, the JTextField is **editable**. If you do not want the user to be able to enter data in a JTextField, you can use the **setEditable() method** to change the editable status of a JTextField. For example, if you want to give a user only one chance to answer a question correctly, then you can prevent the user from replacing or editing the characters in the JTextField by using the code

`answer.setEditable(false);`. If conditions change, and you want the user to be able to edit the JTextField, use the code `answer.setEditable(true);`.

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.

The structure of the JButton class is shown in Figure 9-12.

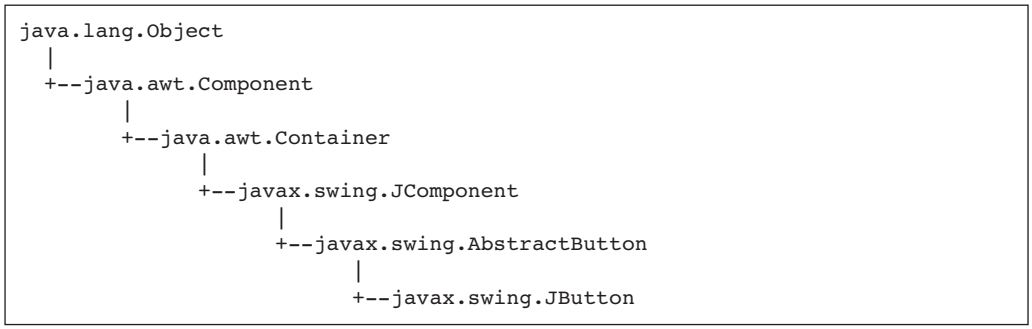

Figure 9-12 JButton class structure

To create a JButton with the label "Press when ready", you write `JButton readyButton = new JButton("Press when ready");`. To add the JButton to an applet, you write `con.add(readyButton);`, where `Container con` has been created using `Container con = getContentPane();`. You can change a JButton's Label with the **setLabel() method**, as in `readyJButton.setLabel("Don't press me again!");`, or get the JLabel and assign it to a String object with the **getLabel() method**, as in `String whatsOnJButton = readyButton.getLabel();`.

Make sure that the label on your JButton describes its function for the user.

As with JTextField components, you can use the `requestFocus()` method with JButton components. The surface of the button that has the keyboard focus appears with an outline so it stands out from the other JButtons.

Adding Multiple Components to a JApplet

Previously you have added only a single component to the Swing applet. In the following example you will add three components. To place multiple components at some given position in a container, you must use a **layout manager** to control component positioning. The normal default behavior of a Swing applet is to use a border layout if no layout manager is specified. **Border layouts**, created by using the BorderLayout class, divide a container into five sections: north, south, east, west, and center. A border layout is created with the **BorderLayout()** or **BorderLayout(int, int) constructor methods**. The statement `BorderLayout border = new BorderLayout();` creates a BorderLayout object named border. The statement `BorderLayout gap = new BorderLayout(5, 10);` creates a BorderLayout object named gap with a horizontal gap of five pixels and a vertical gap of 10 pixels between components. The components in the north, south, east, and west areas will take up as much space as needed; the center will use whatever space is left over.

When a component uses BorderLayout, components are placed in the center region. This means when you add multiple components to a JApplet, the components all lie in the center and appear to be on top of each other, so you can only see the last component you have added.

You will learn more about layout managers in Chapter 14.

When you use a flow layout, 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, it automatically spills components onto the next row. The default positioning of the row of components is centered in the container. In the FlowLayout class there are three row positioning options specified by constants defined in the class. These options are FlowLayout.LEFT, FlowLayout.RIGHT, and FlowLayout.CENTER. To create a layout manager named flow that positions the components to the right, use the statement `FlowLayout flow = new FlowLayout(FlowLayout.RIGHT);`. The layout of a container named con can be set to FlowLayout with row components positioned to the right using `con.setLayout(flow);`. A more compact syntax that combines the two statements into one is `con.setLayout(new FlowLayout(FlowLayout.RIGHT);`. You can use whichever syntax you prefer.

Next you will add a JTextField and a JButton to your Swing applet.

To add a JTextField and a JButton to the JGreet2 JApplet:

1. Open the **JGreet2.java** file in your text editor and change the class name to `JGreet3`.
2. Position the insertion point at the end of the line that defines the bigFont Font object, and then press **[Enter]** to start a new line of text.

3. Declare a JButton with the label "Press Me" and an empty JTextField by typing the following:

```
JButton pressMe = new JButton("Press Me");
JTextField answer = new JTextField("",10);
```

4. Set the new layout manager to a flow layout with the statement: `FlowLayout flow = new FlowLayout();`.
5. Position the insertion point at the end of the statement `con.add(greeting);`, press **[Enter]** to start a new line, and type `con.setLayout(flow);`.
6. Add the JTextField and the JButton to the Swing applet by typing the following:

```
con.add(answer);
con.add(pressMe);
```

7. On the next line, request focus for the answer by typing:
`answer.requestFocus();`
8. Save the file as **JGreet3.java** and compile. Run the applet, changing the TestJGreet2.html document created earlier to **TestJGreet3.html**. Change the class name internally to `JGreet3`, and then execute the `appletviewer TestJGreet3.html` command. Output is shown in Figure 9-13. Confirm that you can type characters into the JTextField and that you can click the JButton using the mouse. You haven't coded any action to take place as a result of a JButton click yet, but the components should function. Also confirm that the objects in each row are positioned to the right.

Figure 9-13 Output of the JGreet3.html document

9. Close the Applet Viewer window.

Learning About Event-Driven Programming

An **event** occurs when someone using your applet takes action on a component, such as clicking the mouse on a JButton object. The programs you have written so far in this book have been **procedural**—you dictated the order in which events occurred. You retrieved user input, wrote decisions and loops, and created output. When you retrieved user input, you had no control over how much time the user took to enter a response

to a prompt, but you did control the fact that processing went no further until the input was completed. In contrast, with **event-driven programs**, 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 moment in 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 italics, 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 can receive all events—you probably have used programs in which clicking many areas of the screen has no effect. If you want an object, such as your applet, to be a listener for an event, you must register the object as a listener for the source.

Newspapers around the world register with news services, such as the Associated Press or United Press International. The news services maintain a list of subscribers, and send each one a story when important national or international events occur. Similarly, a Java component source object (such as a button) maintains a list of registered listeners and notifies all registered listeners (such as an applet) when any event occurs, such as a mouse click. When the listener "receives the news," an event-handling method that is part of the listener object responds to the event.

A source object and a listener object can be the same object. For example, a JButton can change its label when a user clicks it.

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

- Prepare your Swing applet to accept event messages.
- Tell your Swing applet to expect events to happen.
- Tell your Swing applet how to respond to any events that happen.

Preparing Your Swing Applet to Accept Event Messages

You prepare your applet to accept mouse 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**, or a set of specifications for methods that you can use with Event objects. Implementing ActionListener provides you with standard event method specifications that allow your applet to work with ActionEvents, which are the types of events that occur when a user clicks a button.

You can identify interfaces such as ActionListener because they are implemented, and not imported or extended.

Telling Your Swing Applet to Expect Events to Happen

You tell your applet 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, then aButton is the source of a message, and you can think of your applet as a target to which to send a message. You learned in Chapter 4 that the `this` reference means "this current method," so `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 Chapters 13 and 14.

Telling Your Swing Applet How to Respond to Any Events That Happen

The ActionListener interface contains the **actionPerformed(ActionEvent e) method** specification. When a JApplet has registered as a listener with a JButton, and a user clicks the JButton, the actionPerformed() method executes. You must write the actionPerformed() method, which contains a header and a body like all methods. You use the header `public void actionPerformed(ActionEvent e)`, where `e` is any name you choose for the Event (the JButton click) that initiated the notification of the ActionListener (the JApplet). 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 9-14 shows an actionPerformed() method that produces a line of output at the operating system prompt.

```
public void actionPerformed(ActionEvent someEvent)
{
   System.out.println
     ("I'm inside the actionPerformed() method!");
}
```

Figure 9-14 The actionPerformed() method that produces a line of output

When more than one component is added and registered to a Swing applet, it is necessary to determine which component was used for your program to act accordingly. ActionEvent and other event objects are part of the java.awt.event package and are subclasses of the EventObject class.

Every event-handling method is sent an event object of some kind. To determine the source of the event, the getSource() method of the object is used to determine the component that sent the event. For example, if the method header `public void ActionPerformed(ActionEvent e) {` is followed by the statement `Object source = e.getSource();`, the object returned by the getSource() method determines the component that sent the event. Continuing the example, if the source of the event is a JButton named exit, then the following `if` statement evaluates to `true` and the statements contained in its body execute:

```
if(source == exit)
{
//execute these statements
}
```


You can also use the `instanceof` keyword inside an event-handling method 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 any JTextField, regardless of name, generates an action event when text is typed in it and [Enter] is pressed, the following `if` statement would execute:

```
void actionPerformed(ActionEvent e)
{
   Object source = e.getSource();
   if (source instanceof JTextField)
   {
   //execute these statements
   }
}
```

Next you will make your applet an event-driven program by adding functionality to your Swing applet. When the user enters a name and clicks the JButton, the JApplet will display a greeting on the command line.

To add functionality to your Swing applet:

1. Open the **JGreet3.java** file in your text editor and change the class name to `JGreet4`.
2. Add a third import statement to your program by typing: `import java.awt.event.*;`.
3. Position the insertion point at the end of the class header `public class JGreet4 extends JApplet`, press **[Spacebar]**, and then type `implements ActionListener`.

4. Position the insertion point at the end of the statement in the init() method that adds the pressMe button to the JApplet, and press **[Enter]**. Prepare your Swing applet for JButton-sourced events by typing the statement **`pressMe.addActionListener(this);`**.
5. Position the insertion point to the right of the closing curly brace for the init() method, and then press **[Enter]**. Add the following actionPerformed() method which follows the init() method but comes before the closing brace for the JGreet4 class. Use the object's getSource() method to determine that the source of the event is the JButton. Use an `if` statement to control the events that occur when the event's source is the JButton. You will declare a String to hold the user's name, use the getText() method on the answer JTextField to retrieve the String, and display an on-screen message to the user.

```
public void actionPerformed(ActionEvent thisEvent)
{
  Object source = thisEvent.getSource();
  if(source == pressMe)
  {
    String name = answer.getText();
    System.out.println("Hi there " + name);
  }
}
```

6. Save the file as **JGreet4.java**, and compile the program. Edit the file TestJGreet.html and change the class reference to **`TestJGreet4.class`**. Save the file as **TestJGreet4.html**. Run the program using the **`appletviewerTestJGreet4.html`** command.
7. Type your name in the JTextField, and then click the **Press Me** button. Examine your command-prompt screen. The personalized message ("Hi there" and your name) should appear on the command prompt screen.

You might need to drag the Applet Viewer window to a new position so you can see the output on the command line.

8. Drag the mouse to highlight the name in the text field in the Applet Viewer window, and then type a different name. Click the **Press Me** button. A new greeting appears on the command-line screen.
9. Close the Applet Viewer window.

When Swing applets contain a JTextField, there are two ways to get the applet to accept user input. You can enter text and click a button, or you can enter text and press [Enter]. If your Swing applet needs to receive an event message from a JTextField, then you must make your applet a registered Event listener with the JTextField.

To add the ability to press [Enter] from within the JTextField for input:

1. In the JGreet4.java text file, change the class name to **JGreet5**, position the insertion point at the end of the statement `pressMe.addActionListener(this);`, and then press **[Enter]**.
2. Make the answer field accept input by typing: `answer.addActionListener(this);`
3. Add the following statements to the end of the ActionPerformed method that uses the `instanceof` keyword to test for an action event generated by the JTextField named answer:

```
else if(source instanceof JTextField)
     {
       String name = answer.getText();
       System.out.println("Hi there " + name);
     }
```

4. Save the file as **JGreet5.java**, and compile the program. Edit the file TestJGreet4.html and change the class reference to **TextJGreet5.class**. Save the file as **TestJGreet5.html**. Run the program using the **appleviewer TestJGreet5.html** command. To confirm that you can cause the message to appear, type a name and then press **[Enter]**.
5. Close the Applet Viewer window.

Adding Output to a Swing Applet

A Swing applet that produces output on the command-line screen is not very exciting. Naturally, you will want to make changes as various events occur. For example, rather than using `System.out.println("Hi" + name);` to send a greeting to the command-line screen, you might want to add a greeting to the Swing applet itself. One approach is to create a new JLabel that gets added to the applet with the add() method after the user enters a name. You can declare a new, empty JLabel with the statement `JLabel personalGreeting = new Label("");`. After the name is retrieved, you can use the setText() method to set the JLabel text for personalGreeting to `"Hi there " + name`.

To add a personalGreeting JLabel to the Swing applet:

1. Within the JGreet5.java text file, change the class name to **JGreet6**, and then remove both **System.out.println("Hi" + name);** statements from the actionPerformed() method.
2. Position the insertion point at the end of the statement `JTextField answer = new JTextField("",10);` and press **[Enter]**. To declare

a new JLabel named personalGreeting, type the statement:
`JLabel personalGreeting = new JLabel("");`

3. Position the insertion point in the init() method after the `con.add(pressMe);` statement, and then add the JLabel personalGreeting with the statement `con.add(personalGreeting);`, then press **[Enter]**.
4. Add the following statement to the actionPerformed() method after the `String name = answer.getText();` statement to set the text of the personalGreeting. Be sure to add the statement to the body of both the `if` and `if…else` statements.

```
personalGreeting.setText("Hi " + name);
```

5. Save the program as **JGreet6.java**, and compile the file. Edit the file TestJGreet5.html and change the class reference to `TestJGreet6.class`. Save the file as **TestJGreet6.html**. Run the program using the `appletviewer TestJGreet6.html` command. Type a name in the JTextField, and then press **[Enter]** or click the **Press Me** button.
6. Close the Applet Viewer window.

If you can add components to an applet, you should also be able to remove them; you do so with the **remove() method**. For example, after a user enters a name into the JTextField, you might not want the user to use the JTextField or its JButton again, so you can remove them from the applet. To use the remove() method, you place the component's name within the parentheses. As with the add() method, you must redraw the applet after the remove() method to display the effects.

To remove the JTextField and JButton from the Greet applet:

1. Open the **JGreet6.java** file, if necessary, and rename the class `JGreet7`. Place the insertion point after the closing curly brace of the `if…else` statement in the actionPerformed() method, and press **[Enter]**. Then enter the following statements. Note that the repaint() method causes the Swing applet to redraw after the JButton and JTextField are removed from the screen.

```
remove(answer);
remove(pressMe);
repaint();
```

2. Save the file **as JGreet7.java** and compile the file. Edit the file TestJGreet6.html and change the class reference to `TestJGreet.class`. Save the file as **TestJGreet7.html**. Run the program using the `appletviewer TestJGreet7.html` command. Enter a name, and then either press **[Enter]** or click the **Press Me** button. Notice that the JTextField and the JButton disappear from the screen.
3. Close the Applet Viewer window.

Understanding the Swing Applet Life Cycle

Swing applets are popular because they are easy to use in a Web page. Because applets execute in a browser, the JApplet class contains methods that are automatically called by the browser. Earlier in this chapter you learned the names of four of these methods: init(), start(), stop(), and destroy().

You have already written your own init() methods. When you write a method that has the same method header as an automatically provided method, you replace or **override** the original version. Every time a Web page containing a Swing applet is loaded in the browser or when you run the `appletviewer` command within an HTML document that calls a Swing applet, if you have written an init() method for the Swing applet, that method executes; otherwise the automatically provided init() method executes. You should write your own init() method when you have any initialization tasks to perform, such as setting up user interface components.

When you override a method, you create your own version that Java uses, instead of using the automatically supplied version with the same name. It is not the same as overloading a method, which is writing several methods that have the same name but take different arguments. You learned about overloading methods in Chapter 4.

The **start() method** executes after the init() method, and it executes again every time the applet becomes active after it has been inactive. For example, if you run a Swing applet using the `appletviewer` command, and then minimize the AppletViewer window, the Swing applet becomes inactive. When you restore the window, the Swing applet becomes active again. On the Internet, users can leave a Web page, visit another page, and then return to the first site. Again, the Swing applet becomes inactive, and then active. When you write your own start() method you must include any actions you want your Swing applet to take when a user revisits the Swing applet. For example, you might want to resume some animation that you suspended when the user left the applet.

When a user leaves a Web page (perhaps by minimizing a window or traveling to a different Web page) the **stop() method** is invoked. You override the existing empty stop() method only if you want to take some action when a Swing applet is no longer visible. You don't usually need to write your own stop() methods.

The **destroy() method** is called when the user closes the browser or AppletViewer. Closing the browser or AppletViewer releases any resources the Swing applet might have allocated. As with the stop() method, you do not usually have to write your own destroy() methods.

Advanced Java programmers override the stop() and destroy() methods when they want to add instructions to "suspend a thread," or stop a chain of events that were started by a Swing applet, but which are not yet completed.

Again, every Swing applet has the same life cycle outline, as shown in Figure 9-15. When it executes, the init() method runs, followed by the start() method. If the user leaves the Swing applet's page, the stop() method executes. When the user returns, the start() method executes. The stop() and start() sequence might continue any number of times, until the user closes the browser (or AppletViewer) which invokes the destroy() method.

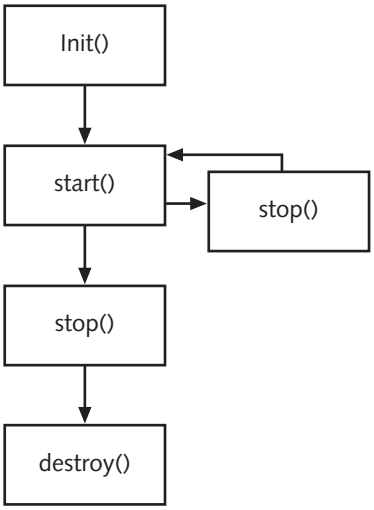

Figure 9-15 Swing applet life cycle

To demonstrate the life cycle methods in action, you can write a Swing applet that overrides all four methods. Note the number of times each method executes.

To demonstrate the life cycle of a Swing applet:

1. Open a new text file in your text editor, and then type the following import statements:

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

2. So the Swing applet will include a JButton that the user can click and, so the ActionListener will be implemented, type the following header for a JLifeCycle applet:

```
public class JLifeCycle extends JApplet implements
  ActionListener
```

3. Press **[Enter]**, type the opening curly brace for the class, and then press **[Enter]** again to start a new line.

4. Declare the following six JLabel objects that you will use to display each name of the six methods that will execute during the lifetime of the Swing applet:

```
JLabel messageInit = new JLabel("init ");
JLabel messageStart = new JLabel("start ");
JLabel messageDisplay = new JLabel("display ");
JLabel messageAction = new JLabel("action ");
JLabel messageStop = new JLabel("stop ");
JLabel messageDestroy = new JLabel("destroy ");
```

5. Declare a JButton by typing: `JButton pressButton = new JButton("Press");`.

6. Declare six integers that will hold the number of occurrences of each of the six methods by typing the following code on one line:

```
int countInit, countStart, countDisplay, countAction,
countStop, countDestroy;
```

7. Start the init() method by adding a container and flow layout manager with the statements:

```
public void init()
{
  Container con = getContentPane();
  con.setLayout (new FlowLayout());
```

8. Add the following statements, which adds one to countInit, places the components within the applet, and then calls the display() method:

```
  ++countInit;
  con.add(messageInit);
  con.add(messageStart);
  con.add(messageDisplay);
  con.add(messageAction);
  con.add(messageStop);
  con.add(messageDestroy);
  con.add(pressButton);
  pressButton.addActionListener(this);
  display();
}
```

9. Add the following start() method, which adds one to countStart and calls display():

```
public void start()
{
  ++countStart;
  display();
}
```

10. Add the following display() method, which adds one to countDisplay, displays the name of each of the six methods with the current count, and indicates how many times the method has executed:

```
public void display()
{
  ++countDisplay;
  messageInit.setText("init " + countInit);
  messageStart.setText("start " + countStart);
  messageDisplay.setText("display " + countDisplay);
  messageAction.setText("action " + countAction);
  messageStop.setText("stop " + countStop);
  messageDestroy.setText("destroy " + countDestroy);
}
```

11. Add the following stop() and destroy() methods, which each add one to the appropriate counter and call display():

```
public void stop()
{
  ++countStop;
  display();
}
public void destroy()
{
  ++countDestroy;
  display();
}
```

12. When the user clicks pressButton, the following actionPerformed() method will execute; it adds one to countAction and displays it. Enter the method:

```
public void actionPerformed(ActionEvent e)
{
  Object source = e.getSource();
  if(source == pressButton)
  {
    ++countAction;
    display();
  }
}
```

13. Add the closing curly brace for the class. Save the program as **JLifeCycle.java** in the Chapter.09 folder on your Student Disk. If necessary, compile, correct any errors, and compile again.

Take a moment to examine the code you created for JLifeCycle.java. Each method adds one to one of the six counters, but you never explicitly call any of the methods except display(); each of the other methods will be called automatically. Next you will create an HTML document so you can test JLifeCycle.java.

To create an HTML document to test JLifeCycle.java:

1. Open a new text file in your text editor.
2. Enter the following HTML code:

```
<HTML>
<APPLET CODE="JLifeCycle.class" WIDTH = 460 HEIGHT = 200>
</APPLET>
</HTML>
```

3. Save the file as **TestJLifeCycle.html** in the Chapter.09 folder on your Student Disk.
4. Run the HTML document using the command **`appletviewer TestJLifeCycle.html`**. Figure 9-16 shows the output. When the applet begins, the init() method is called, so one is added to countInit. The init() method calls display(), so one is added to countDisplay. Immediately after the init() method executes, the start() method is executed, and one is added to countStart. The start() method calls display(), so one more is added to countDisplay. The first time you see the applet, countInit is 1, countStart is 1, and countDisplay is 2. The methods actionPerformed(), stop(), and destroy() have not yet been executed.

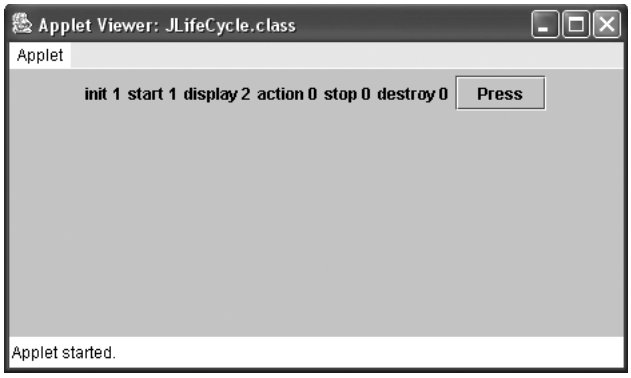

Figure 9-16 JLifeCycle Swing applet after start-up

5. Click the **Minimize** button to minimize the Applet Viewer window, and then click the **Taskbar** button to restore it. The applet now looks like Figure 9-17. The init() method still has been called only once, but when you minimized the applet, the stop() method executed, and when you restored it, the start() method executed. Therefore, countStop is now 1 and countStart has increased to 2. Additionally, because start() and stop() call display(),countDisplay() is increased by two and now holds the value 4.

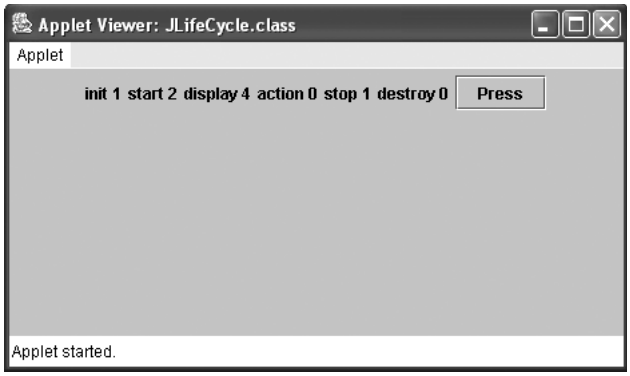

Figure 9-17 JLifeCycle Swing applet after being minimized and restored

6. Minimize and maximize the Applet Viewer window again. Now the stop() method has executed twice, the start() method has executed three times, and the display() method has executed a total of six times, as shown in Figure 9-18.

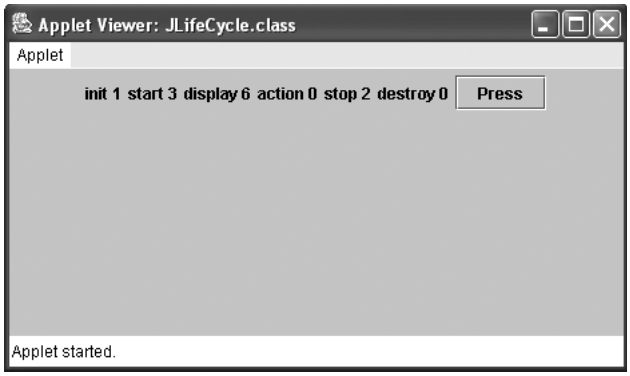

Figure 9-18 JLifeCycle Swing applet after being minimized and restored twice

7. Click the **Press** button. The count for the actionPerformed() method is now 1, and actionPerformed() calls display(), so countDisplay() is now 7, as shown in Figure 9-19.
8. Continue to minimize, maximize, and click the **Press** button. Note the changes that occur with each activity until you can correctly predict the outcome. Notice that the destroy() method is not executed until you close the applet, and then it is too late to observe an increase in countDestroy.

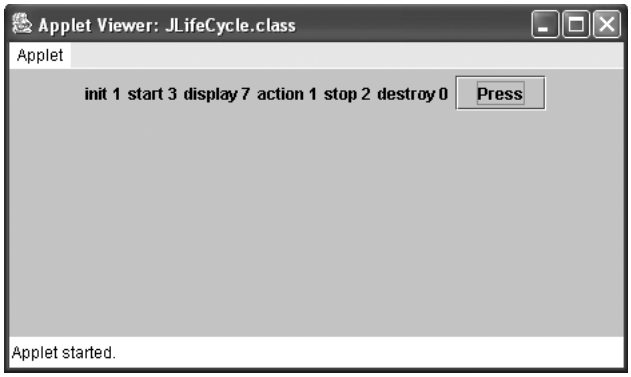

Figure 9-19 JLifeCycle Swing applet after clicking the Press button

Creating a More-Sophisticated Interactive Swing Applet

You are now able to create a fairly complex application or applet. Next you will create an applet that contains several components, receives user input, makes decisions, uses arrays, performs output, and reacts to the applet life cycle.

The JPartyPlanner Swing applet lets its user estimate the cost of an event hosted by Event Handlers Incorporated. Event Handlers uses a sliding fee scale so the per-guest cost decreases as the total number of invited guests increases. Table 9-1 shows the fee structure.

Table 9-1 Cost per guest for events

Number of Guests	Cost per Guest
1 to 24	$27
25 to 49	$25
50 to 99	$22
100 to 199	$19
200 to 499	$17
500 to 999	$14
1000 and over	$11

The Swing applet lets the user enter a number of anticipated guests. The user can press [Enter] or click a JButton to perform the fee lookup and event cost calculation. Then the Swing applet displays the cost per person as well as the total cost for the event. The user can continue to request fees for a different number of guests and view the results for any length of time before making another request or leaving the page. When the user leaves the page, however, you will erase the last number of requested guests and ensure that the next user starts fresh with zero guests.

To begin creating an interactive party planner Swing applet:

1. Open a new text file in your text editor.
2. Type the following import statements, the JPartyPlanner class header, and the opening curly brace for the class:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JPartyPlanner extends JApplet implements
  ActionListener
{
```

3. You will need several components: a JLabel for the company name, a JButton the user can click to perform a calculation, and two more JLabels to display output. Add the following code to implement these components:

```
JLabel companyName = new
  JLabel("Event Handlers Incorporated");
JButton calcButton = new JButton("Calculate");
JLabel perPersonResult = new JLabel("Plan with us.");
JLabel totalResult = new JLabel("The more the merrier");
```

4. To enhance the appearance, create a Font object by typing the following:

```
Font bigFont = new Font("Helvetica", Font.ITALIC, 24);
```

5. Use the init() method to place components within the applet screen, and then prepare the JButton to receive action messages by typing the following:

```
public void init()
{
Container con = getContentPane();
con.setLayout(new FlowLayout());
companyName.setFont(bigFont);
con.add(companyName);
con.add(calcButton);
calcButton.addActionListener(this);
con.add(perPersonResult);
con.add(totalResult);
}
```

6. Add the following start() method, which executes when the user leaves the Swing applet and resets the JLabel and the data-entry JTextField:

```
public void start()
{
  perPersonResult.setText("Plan with us.");
  totalResult.setText("The more the merrier");
  repaint();
}
```

7. Save the partially completed Swing applet as **JPartyPlanner.java** in the Chapter.09 folder on your Student Disk.

You finished the init() and start() methods for the JPartyPlanner Swing applet, placed each component in the Swing applet, and reinitialized each component every time a user returns to the Swing applet after leaving. The Swing applet doesn't actually do anything yet; most of the applet's work will be contained in the actionPerformed() method, the most complicated method in this applet.

Next you will create the actionPerformed() method. You begin by declaring two parallel arrays—one array will hold guest limits for each of six event rates, and the other array will hold the actual rates.

To complete the JPartyPlanner Swing applet:

1. Use a dialog box to receive user input for the number of guests. Enter the following method header for actionPerformed() and declare two arrays for guest limits and rates:

```
public void actionPerformed(ActionEvent e)
{
Object source = e.getSource();
  if(source == calcButton)
  {
    String response = JOptionPane.showInputDialog(null,
      "Enter the number of guests");
    int[] guestLimit =   {0, 25, 50, 100, 200, 500, 1000};
    int[] ratePerGuest = {27, 25, 22, 19, 17, 14, 11};
```

2. Next add the following variable to hold the number of guests. The user will receive input from a dialog box, but you need an integer to perform calculations so you can use the parseInt() method.

```
int guests = Integer.parseInt(response);
```

You learned about the parseInt() method in Chapter 6.

3. You need two variables—one will hold the individual, per-person fee for an event, and the other will hold the fee for the entire event. Enter the following variables:

```
int individualFee = 0, eventFee = 0;
```

4. Enter the following variables to use as subscripts for the arrays:

```
int x = 0, a = 0;
```

There are several ways to search through the guestLimit array to discover the appropriate position of the per person fee in the ratePerGuest array. One possibility is to use a `for` loop and decrement from six down to zero. If the number of guests is greater than or equal to any value in the guestLimit

array, then the corresponding per person rate in the ratePerGuest array is the correct rate. After finding the correct individual rate, you determine the price for the entire event by multiplying the individual rate by the number of guests. After finding the appropriate individual fee for a given event, you no longer want to search through the guestLimit array, so you set the subscript x equal to zero to force an early exit from the `for` loop.

5. Enter the following `for` loop:

```
for(x = 6; x >= 0; --x)
  if(guests >= guestLimit[x])
    {
      individualFee = ratePerGuest[x];
      eventFee = guests * individualFee;
      x = 0;
    }
```

6. The only tasks that still must be included in the actionPerformed() method involve producing output for the user. Enter the following code to accomplish this processing:

```
perPersonResult.setText("$" + individualFee + " per
  person");
totalResult.setText("Event cost $" + eventFee);
```

7. Add three closing curly braces—two for the actionPerformed() method, and one for the entire JPartyPlanner Swing applet.

8. Save the file, and then compile it at the command prompt.

9. Open a new text file in your text editor, and then create the following HTML document to test the applet:

```
<HTML>
<APPLET CODE="JPartyPlanner.class"
  WIDTH = 320 HEIGHT = 200>
</APPLET>
</HTML>
```

10. Save the HTML document as **TestJPartyPlan.html** in the Chapter.09 folder on your Student Disk. Then use the **`appletviewer`** command to execute the file. Test the applet with different numbers of guests until you are sure that the per person rates and event rates are correct. Minimize and restore the Applet Viewer window and observe that any calculated fees are replaced with start() messages. For example, if you enter 100 guests, then your output should resemble Figure 9-20.

11. Close the Applet Viewer window.

Figure 9-20 Output of JPartyPlanner Swing applet

USING THE SETLOCATION() AND SETENABLED() METHODS

A serious shortcoming of the objects you have written so far is that you cannot choose the location of the JLabel and JButton objects you place within your Swing applets. When you use the add() method to add a component to an applet, it seems to have a mind of its own as to where it is physically placed. Although you must learn more about the Java programming language before you can change the initial placement of components when you use the add() method, you can use the setLocation() method to change the location of a component at a later time. The **setLocation() method** allows you to place a component at a specific location within the Applet Viewer window.

Any applet window consists of a number of horizontal and vertical pixels on the screen. You set the pixel values in the HTML document you write to test the Swing applet. Any component you place on the screen has a horizontal, or **x-axis**, position as well as a vertical, or **y-axis**, position in the window. The upper-left corner of any display is position 0,0. The first, or **x-coordinate**, value increases as you travel from left to right across the window. The second, or **y-coordinate**, value increases as you travel from top to bottom.

For example, to position a Label object named someLabel at the upper-left corner of a window, you write `someLabel.setLocation(0,0);`. If a window is 200 pixels wide and 100 pixels tall, then you can place a Button named pressMe in the approximate center of the window with the statement `pressMe.setLocation(100,50);`. Figure 9-21 illustrates the screen coordinate positions.

You can picture a coordinate as an infinitely thin line that lies between the pixels of the output device.

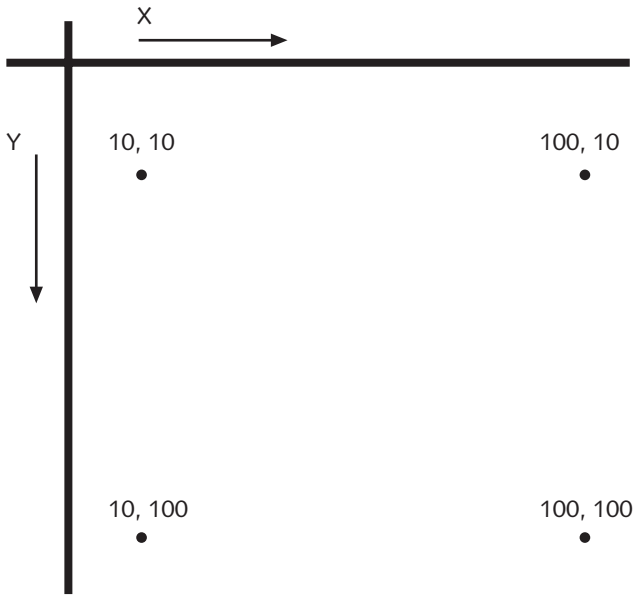

Figure 9-21 Screen coordinate positions

When you use setLocation(), the upper-left corner of the component is placed at the specified x- and y-coordinates. If a window is 100 by 100 pixels, then `aButton.setLocation(100,100);` places the JButton outside the window, where you cannot see the component.

Next you will create a JLabel that changes its location with each JButton click.

To create a moving JLabel:

1. Open a new text file in your text editor, and then type the following import statements:

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

2. Type the following class header and opening curly brace for a class named JMoveLabel. You must implement ActionListener because the Swing applet requires that the user click a JButton as the action event.

```
public class JMoveLabel
  extends JApplet implements ActionListener
```

3. Declare the following JLabel, JButton, and two integers that will hold the horizontal and vertical coordinates of the JLabel:

```
JLabel movingMsg = new JLabel("Event Handlers Inc.");
JButton pressButton = new JButton("Press");
int xLoc = 20, yLoc =20;
```

4. Enter the following init() method to add the components to the Swing applet screen and prepare the JButton to receive messages:

```
public void init()
{
  Container con = getContentPane();
  con.setLayout(new FlowLayout());
  con.add(movingMsg);
  con.add(pressButton);
  pressButton.addActionListener(this);
}
```

5. Enter the following actionPerformed method() to move the message 10 pixels to the right and 10 pixels down each time the user clicks the JButton:

```
public void actionPerformed(ActionEvent e)
{
  Object source = e.getSource();
  if(source == pressButton)
     movingMsg.setLocation(xLoc+=10, yLoc+=10);
}
```

6. Add the closing curly brace to the class.
7. Save the file as **JMoveLabel.java** in the Chapter.09 folder on your Student Disk, and then compile it.
8. Open a new file in your text editor, and then create the following HTML document to test the Swing applet:

```
<HTML>
<APPLET CODE="JMoveLabel.class" WIDTH = 460 HEIGHT = 300>
</APPLET>
</HTML>
```

9. Save the HTML document as **Test JMoveLabel.html** in the Chapter.09 folder on your Student Disk. Then run the file using the `appletviewer TestJMoveLabel.html` command. Click the **pressButton** and observe how the JLabel moves each time you click it. If you click enough times, the JLabel moves off the screen.
10. Close the Applet Viewer window.

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. You can use the **setEnabled() method** with a component to make it unavailable and, in turn, to make it available again. The setEnabled() method takes an argument of `true` if you want to enable a component, or `false` if you want to disable a component.

When you create a component, it is enabled by default.

For example, in the JMoveLabel applet, a user can continue to click the JButton until the JLabel moves completely off the screen. If you want to prevent this from happening, you can disable the JButton after the JLabel has advanced as far as you want it to go. Next you will stop the JLabel from moving after it reaches a y-coordinate of 280.

To disable the JButton:

1. Open the **JMoveLabel.java** file in your text editor and change the class name to **JMoveLabel2**.
2. Position the insertion point at the end of the `if` statement in the actionPerformed() method, and then press **[Enter]** to start a new line of text. Add the following statement to disable the JButton when the message has moved to a y-coordinate of 280:

```
if(yLoc==280)
  pressButton.setEnabled(false);
```

3. Save the file as **JMoveLabel2.java** and compile. Change the Test JMoveLabel.html document created earlier to **Test JMoveLabel2.html**, change the class internally to `JMoveLabel2.class`, and then type the `appletviewer TestJMoveLabel2.html` command to run the applet. Click **pressButton** until the JButton is disabled and the JLabel cannot descend any farther.
4. Close the Applet Viewer window.

Chapter Summary

- Swing applets are programs that are called from within another application. You run them within a Web page, or within another program called Applet Viewer, which comes with the Java Developer's Kit. An applet must be called from within an HTML (Hypertext Markup Language) document.
- A component is not added directly to a Swing applet. Instead, you use the add() method to add a component to a container.
- Every Swing applet includes four methods: `public void init()`, `public void start()`, `public void stop()`, and `public void destroy()`.
- 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 inputs the data by pressing [Enter] on the keyboard or clicking a JButton with the mouse. The setText() method allows you to change the text in a previously created TextField. The getText() method allows you to retrieve the String of text in a TextField.

- You can create a JButton with or without a label. You can change a JButton's label with the setLabel() method, or get the JLabel and assign it to a String object with the getLabel() method.
- JLabel is a built-in class that holds text that can be displayed within a Swing applet. The setText() method assigns text to a JLabel or any other component.
- An event occurs when a Swing applet's user takes action on a component, such as using the mouse to click a JButton object. In event-driven programs, the user might initiate any number of events in any order. 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.
- To respond to user events within any Swing applet you create, you must prepare your applet to accept event messages, tell your applet to expect events to happen, and then tell your applet how to respond to any events that happen. Adding `implements ActionListener` to an applet's class header prepares a Swing applet to receive event messages.
- An ActionEvent is the type of event that occurs when a user clicks a JButton. You tell an applet to expect ActionEvents with the addActionListener() method. The ActionListener interface contains the actionPerformed(ActionEvent e) method specification. In the body of the method, you write any statements that you want to execute when an action takes place.
- Every event-handling method is sent an event of some kind. The object's getSource() method can be used to determine the component that sent the event.
- When you use the add() method to add a component to an applet, you do not specify the physical location of the component in the Applet Viewer window. The setLocation() method allows you to place a component at a specific location within an Applet Viewer window. When you include x- and y-coordinates within the setLocation() method, the upper-left corner of the component is placed at the specified location.
- You can use the setEnabled() method with a component to make it unavailable and, in turn, to make it available again. The setEnabled() method takes an argument of `true` if you want to enable a component, or `false` if you want to disable a component.

Review Questions

1. A major difference between AWT and Swing applets is _____.
   a. the AWT applet uses a content pane
   b. they are executed using different Java commands
   c. they are executed from within different HTML documents
   d. the Swing applet imports from the .javax.swing package

2. A program that allows you to display HTML documents on your computer screen is a _____.
   a. search engine
   b. compiler
   c. browser
   d. server
3. The name of any Swing applet called using CODE within an HTML document must use the _____ extension.
   a. .exe
   b. .code
   c. .java
   d. .class
4. The _____ is a String representing a font.
   a. point size
   b. style
   c. leading
   d. typeface
5. A JTextField is a Swing component _____.
   a. into which a user can type a single line of text data
   b. into which a user can type multiple lines of text data
   c. that automatically has focus when the applet runs
   d. whose text cannot be changed
6. The Swing add() method _____.
   a. adds two integers
   b. adds a component directly to the Swing applet
   c. places a component within a container
   d. places a text value within an applet component
7. The start() method called in any Swing applet is called _____.
   a. at start-up
   b. when the user closes the browser
   c. when a user revisits an applet
   d. when a user leaves a Web page

8. A Font object contains all of the following arguments except _____.
   a. language
   b. typeface
   c. style
   d. point size
9. To respond to user events within a Swing applet, you must _____.
   a. prepare the applet to accept event messages
   b. import the java.applet.* package
   c. tell your applet how to respond to any events that happen
   d. accomplish both a and c
10. The constructor `public JButton("4")` creates _____.
    a. an unlabeled JButton
    b. a JButton four pixels wide
    c. a JButton four characters wide
    d. a JButton with a "4" on it
11. An event occurs when a _____.
    a. component requests focus
    b. component is enabled
    c. component sets text
    d. button is clicked
12. ActionListener is an example of a(n) _____.
    a. import
    b. applet
    c. interface
    d. component
13. When a Swing applet is registered as a listener with a JButton, if a user clicks the JButton, the method that executes is _____.
    a. buttonPressed()
    b. addActionListener()
    c. start()
    d. actionPerformed()

14. When you write a method that has the same method header as an automatically provided method, you _____ the original version.
    a. destroy
    b. override
    c. call
    d. copy
15. Which of the following statements creates a JLabel that says "Welcome"?
    a. `JLabel = new JLabel("Welcome");`
    b. `JLabel aLabel = JLabel("Welcome");`
    c. `aLabel = new JLabel("Welcome");`
    d. `JLabel aLabel = new JLabel("Welcome");`
16. Which of the following statements correctly creates a Font object?
    a. `Font aFont = new Font("TimesRoman", Font.ITALIC, 20);`
    b. `Font aFont = new Font(30, "Helvetica", Font.ITALIC);`
    c. `Font aFont = new Font(Font.BOLD,"Helvetica", 24);`
    d. `Font aFont = new Font(22, Font.BOLD,"TimesRoman");`
17. The method that positions a component within an applet is _____.
    a. position()
    b. setPosition()
    c. location()
    d. setLocation()
18. In a window that is 200 x 200 pixels, position 10, 190 is nearest to the _____ corner.
    a. upper-left
    b. upper-right
    c. lower-left
    d. lower-right
19. An object's _____ method can be used to determine the component that sends an event.
    a. getSource()
    b. instanceof()
    c. both of the above
    d. none of the above

20. Which of the following statements disables a component named someComponent?
    a. `someComponent.setDisabled();`
    b. `someComponent.setDisabled(true);`
    c. `someComponent.setEnabled(false);`
    d. `someComponent.setEnabled(true);`

EXERCISES

1. Create a Swing applet with a JButton labeled "Who's number one?". When the user clicks the button, display your favorite team in a large font. Save the program in the Chapter.09 folder on your Student Disk as **JNumberOne.java**.
2. a. Create a Swing applet that asks a user to enter a password into a JTextField and to then press [Enter]. Compare the password to "Rosebud"; if it matches, display "Access Granted"; if not, display "Access Denied". Save the program in the Chapter.09 folder on your Student Disk as **JPasswordA.java**.
   b. Modify the password applet in Exercise 2a to ignore differences in case between the typed password and "Rosebud". Save the program in the Chapter.09 folder on your Student Disk as **JPasswordB.java**.
   c. Modify the password applet in Exercise 2b to compare the password to a list of five valid passwords: "Rosebud", "Redrum", "Jason", "Surrender", or "Dorothy". Save the program in the Chapter.09 folder on your Student Disk as **JPasswordC.java**.
3. Create a Swing applet with a JButton. When the user clicks the JButton, change the font typeface and style. Save the program in the Chapter.09 folder on your Student Disk as **JChangeFont.java**.
4. Create a Swing applet that displays the date and time in a JTextField with the JLabel "Today is" when the user clicks a JButton. Save the program in the Chapter.09 folder on your Student Disk as **JDayOfYear.java**.
5. Create a Swing applet that displays an employee's title in a JTextField when the user types the employee's first and last names (separated by a space) in another JTextField. Include JLabels for each JTextField. You can use arrays to store the employees' names and titles. Save the program in the Chapter.09 folder on your Student Disk as **JEmployeeTitle.java**.
6. Create a Swing applet that displays an employee's title in a TextField when the user types an employee's first and last names (separated by a space) in another JTextField, or displays an employee's name in a JTextField when the user types the employee's title in a JTextField. Include a JLabel for each JTextField. Add a JLabel at the top of the applet with the text "Enter a name or a title". You can use arrays to store the employees' names and titles. Save the program in the Chapter.09 folder on your Student Disk as **JEmployeeTitle2.java**.

7. Create a Swing applet that uses a dialog box to enter an integer. When the user clicks a JButton, the user is prompted to enter an integer. When the user clicks the OK button, the integer is doubled and the answer is displayed. Save the program in the Chapter.09 folder on your Student Disk as **JDialogDouble.java**.
8. Create a Swing applet that allows the user to enter two integers into two separate dialog boxes. When the user clicks a JButton, the sum of the integers is displayed. Save the program in the Chapter.09 folder on your Student Disk as **JDialogAdd.java**.
9. a. Create an applet named DivideTwo that allows the user to enter two integers in two separate JTextFields. The user can click a JButton to divide the first integer by the second integer and display the result.

   b. Modify the DivideTwo applet created in Exercise 9a so that if a user enters zero for the second integer, when the user clicks the JButton to divide, the Swing applet displays the message "Division by zero not allowed!" Save the final program in the Chapter.09 folder on your Student Disk as **JDivideMe.java**.
10. a. Create a payroll Swing applet named CalcPay that allows the user to enter two double values—hours worked, and an hourly rate. When the user clicks a JButton, gross pay is calculated. Save the program in the Chapter.09 folder on your Student Disk as **JCalculatePay.java**.

    b. Modify the payroll Swing applet created in Exercise 10a so that federal withholding tax is subtracted from gross pay based on the following table:

Income$	Withholding%
0 to 99.99	10
100.00 to 299.99	15
300.00 to 599.99	21
600.00 and up	28

    Save the program in the Chapter.09 folder on your Student Disk as **JCalculatePay2.java**.
11. Create a conversion Swing applet that prompts the user to enter a distance in miles in a dialog box, then converts miles to kilometers and displays the result in a JTextField as "XX.XX kilometers", where XX.XX is the number of kilometers. You can use the formula miles *1.6 to convert miles to kilometers. Save the program in the Chapter.09 folder on your Student Disk as **JConversion.java**.
12. Create a Swing applet that calculates the current balance in a checking account in a JTextField. The user enters the beginning balance, check amount, and deposit amount in separate JTextFields with the appropriate JLabels. After the applet calculates the current balance, reposition the JTextFields and JLabels so that the beginning balance appears on the first line, the check and deposit amounts appear on the second line, and the new balance appears on the third line. Save the program in the Chapter.09 folder on your Student Disk as **JCalculateBalance.java**.

13. Create a Swing applet that displays two of your family members' names, relationships to yourself, and ages, in JTextFields when you click a JButton. Each JTextField should have a JLabel. After clicking the JButton, reposition the JTextFields and JLabels so that your family members' names appear on the second line, and the family members' relationships to you and ages appear on the third line. Save the program in the Chapter.09 folder on your Student Disk as **JFamilyRecord.java**.
14. Each of the following files in the Chapter.09 folder on your Student Disk has syntax and/or logical 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, DebugNine1.java will become FixDebugNine1.java. You can test each applet with the TestDebugNine1.html throughTestDebugNine4.html files on your Student Disk.
    a. DebugNine1.java
    b. DebugNine2.java
    c. DebugNine3.java
    d. DebugNine4.java

CASE PROJECT

Ray's Appliance Store sells a wide variety of kitchen appliances. Customers often ask for an estimate of the annual cost of running an appliance. Ray typically scribbles his calculations with a pencil on a notepad when he can find one. He has asked you to write a Swing applet that will do the calculations on his computer. Ray wants to be prompted to enter the cost per kilowatt hour of electricity and the estimated number of hours the appliance will run annually. After the figures are entered, the applet should display the estimated annual cost.