Module 14

Applets, Events, and Miscellaneous Topics

CRITICAL SKILLS

14.1	Understand applet basics
14.2	Know the applet architecture
14.3	Create an applet skeleton
14.4	Initialize and terminate applets
14.5	Repaint applets
14.6	Output to the status window
14.7	Pass parameters to an applet
14.8	Know the **Applet** class
14.9	Understand the delegation event model
14.10	Use the delegation event model
14.11	Know the remaining Java keywords

Teaching the elements of the Java language is the primary goal of this book, and in this regard, we are nearly finished. The preceding 13 modules have focused on the features of Java defined by the language, such as its keywords, syntax, block structure, type conversion rules, and so on. At this point, you have enough knowledge to write sophisticated, useful Java programs. However, there is an important part of Java programming that requires more than just an understanding of the language itself: *the applet*. The applet is the single most important type of Java application, and no book on Java would be complete without coverage of it. Therefore, this module presents an overview of applet programming.

Applets use a unique architecture and require the use of several special programming techniques. One of these techniques is *event handling*. Events are the way that an applet receives input from the outside world. Since event handling is an important part of nearly all applets, it is also introduced here.

Be forewarned: The topics of applets and event handling are very large. Full and detailed coverage of them is well beyond the scope of this book. Here you will learn their fundamentals and see several examples, but we will only scratch the surface. After finishing this module, however, you will have a solid foundation upon which to begin an in-depth study of these important topics.

This module ends with a description of a few of Java's keywords, such as **instanceof** and **native**, that have not been described elsewhere in this book. These keywords are used for more advanced programming, but they are summarized here for completeness.

CRITICAL SKILL 14.1 Applet Basics

Applets differ from the type of programs shown in the preceding modules. As mentioned in Module 1, applets are small programs that are designed for transmission over the Internet and run within a browser. Because Java's virtual machine is in charge of executing all Java programs, including applets, applets offer a secure way to dynamically download and execute programs over the Web. Before discussing any theory or details, let's begin by examining a simple applet. It performs one function: It displays the string "Java makes applets easy." inside a window.

```
// A minimal applet.
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Java makes applets easy.", 20, 20);
  }
}
```

Notice these **import** statements. They are used by all applets.

This outputs to the applet's window.

This applet begins with two **import** statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical interface. As you might expect, it is quite large and sophisticated. A complete discussion of it would require a book of its own. Fortunately, since we will be creating only very simple applets, we will make only limited use of the AWT. The next **import** statement imports the **applet** package. This package contains the class **Applet**. Every applet that you create must be a subclass of **Applet**.

The next line in the program declares the class **SimpleApplet**. This class must be declared as **public** because it will be accessed by outside code.

Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT **Component** class (which is a superclass of **Applet**) and must be overridden by the applet. **paint( )** is called each time the applet must redisplay its output. This can occur for several reasons. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or the applet window can be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Inside **paint( )**, there is a call to **drawString( )**, which is a member of the **Graphics** class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output beginning at *x*,*y*. In a Java window, the upper-left corner is location 0,0. The call to **drawString( )** in the applet causes the message to be displayed beginning at location 20,20.

Notice that the applet does not have a **main( )** method. Unlike the programs shown earlier in this book, applets do not begin execution at **main( )**. In fact, most applets don't even have a **main( )** method. Instead, an applet begins execution when the name of its class is passed to a browser or other applet-enabled program.

After you have entered the source code for **SimpleApplet**, you compile in the same way that you have been compiling programs. However, running **SimpleApplet** involves a different process. There are two ways in which you can run an applet: inside a browser or with a special development tool that displays applets. The tool provided with the standard Java JDK is called **appletviewer**, and we will use it to run the applets developed in this module. Of course, you can also run them in your browser, but the **appletviewer** is much easier to use during development.

To execute an applet (in either a Web browser or the **appletviewer**), you need to write a short HTML text file that contains the appropriate APPLET tag. (You can also use the newer

OBJECT tag, but this book will use APPLET because this is the traditional approach.) Here is the HTML file that will execute **SimpleApplet**:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

The **width** and **height** statements specify the dimensions of the display area used by the applet.

To execute **SimpleApplet** with an applet viewer, you will execute this HTML file. For example, if the preceding HTML file is called **StartApp.html**, then the following command line will run **SimpleApplet**:

```
C:\>appletviewer StartApp.html
```

Although there is nothing wrong with using a stand-alone HTML file to execute an applet, there is an easier way. Simply include a comment near the top of your applet's source code file that contains the APPLET tag. If you use this method, the **SimpleApplet** source file looks like this:

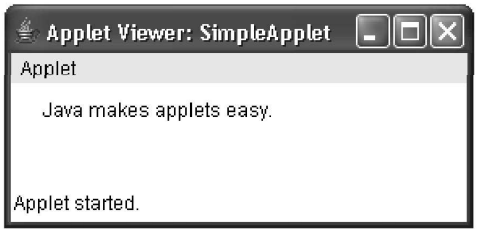
```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("Java makes applets easy.", 20, 20);
  }
}
```

This HTML is used by **appletviewer** to run the applet.

Now you can execute the applet by passing the name of its source file to **appletviewer**. For example, this command line will now display **SimpleApplet**.

```
C:>appletviewer SimpleApplet.java
```

The window produced by **SimpleApplet**, as displayed by **appletviewer**, is shown in the following illustration:



When using **appletviewer**, keep in mind that it provides the window frame. Applets run in a browser will not have a visible frame.

Let's review an applet's key points:

- All applets are subclasses of **Applet**.
- Applets do not need a **main( )** method.
- Applets must be run under an applet viewer or a Java-compatible browser.
- User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by the AWT.

Progress Check

1. What is an applet?
2. What method outputs to the applet's window?
3. What package must be included when creating an applet?
4. How are applets run?

1. An applet is a special type of Java program that is designed for transmission over the Internet and that runs inside a browser.
2. The **paint( )** method displays output in an applet's window.
3. The package **java.applet** must be included when creating an applet.
4. Applets are executed by a browser or by special tools, such as **appletviewer**.

Ask the Expert

Q: **I have heard about something called *Swing*. What is it and how does it relate to the AWT?**

A: Swing is a set of classes that provides powerful and flexible alternatives to the standard AWT components. For example, in addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables. Even familiar components such as buttons have more capabilities in Swing. For these reasons, many Java programmers use Swing when creating applets.

Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term *lightweight* is used to describe such elements.

Although the Swing components offer alternatives to those supplied by the AWT, it is important to understand that Swing is built upon the foundation of the AWT. Thus, a firm understanding of the AWT is required to use Swing effectively.

Applet Organization and Essential Elements

Although the preceding applet is completely valid, such a simple applet is of little value. Before you can create useful applets, you need to know more about how applets are organized, what methods they use, and how they interact with the run-time system.

CRITICAL SKILL 14.2

The Applet Architecture

An applet is a window-based program. As such, its architecture is different from the console-based programs shown in the first part of this book. If you are familiar with Windows programming, you will be right at home writing applets. If not, then there are a few key concepts you must understand.

First, applets are event driven, and an applet resembles a set of interrupt service routines. Here is how the process works. An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return control to the system. This is a crucial point. For the most part, your applet should not enter a "mode" of operation, in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for

example, displaying a scrolling message across its window), you must start an additional thread of execution.

Second, it is the user who initiates interaction with an applet—not the other way around. In a console-based program, when the program needs input, it will prompt the user and then call some input method. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks a mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated. Applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

While the architecture of an applet is not as easy to understand as that of a console-based program, Java makes it as simple as possible. If you have written programs for Windows, you know how intimidating that environment can be. Fortunately, Java provides a much cleaner approach that is more quickly mastered.

CRITICAL SKILL 14.3

A Complete Applet Skeleton

Although **SimpleApplet** shown earlier is a real applet, it does not contain all of the elements required by most applets. Actually, all but the most trivial applets override a set of methods that provide the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods—**init( )**, **start( )**, **stop( )**, and **destroy( )**—are defined by **Applet**. The fifth method, **paint( )**, you have already seen and is inherited from the AWT **Component** class. Since default implementations for all of these methods are provided, applets do not need to override those methods they do not use. These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
  // Called first.
  public void init() {
    // initialization
  }

  /* Called second, after init().  Also called whenever
```

```
     the applet is restarted. */
  public void start() {
    // start or resume execution
  }

  // Called when the applet is stopped.
  public void stop() {
    // suspends execution
  }

  /* Called when applet is terminated.  This is the last
     method executed. */
  public void destroy() {
    // perform shutdown activities
  }

  // Called when an applet's window must be restored.
  public void paint(Graphics g) {
    // redisplay contents of window
  }
}
```

Although this skeleton does not do anything, it can be compiled and run. Thus, it can be used as a starting point for applets that you create.

CRITICAL SKILL 14.4

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are executed. When an applet begins, the following methods are called in this sequence:

1. **init( )**
2. **start( )**
3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**
2. **destroy( )**

Let's look more closely at these methods.

The **init( )** method is the first method to be called. In **init( )** your applet will initialize variables and perform any other startup activities.

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped, such as when the user returns to a previously displayed Web page that contains an applet. Thus, **start( )** might be called more than once during the life cycle of an applet.

The **paint( )** method is called each time your applet's output must be redrawn and was described earlier.

When the page containing your applet is left, the **stop( )** method is called. You will use **stop( )** to suspend any child threads created by the applet and to perform any other activities required to put the applet in a safe, idle state. Remember, a call to **stop( )** does not mean that the applet should be terminated because it might be restarted with a call to **start( )** if the user returns to the page.

The **destroy( )** method is called when the applet is no longer needed. It is used to perform any shutdown operations required of the applet.

Progress Check

1. What are the five methods that most applets will override?
2. What must your applet do when **start( )** is called?
3. What must your applet do when **stop( )** is called?

CRITICAL SKILL 14.5

Requesting Repainting

As a general rule, an applet writes to its window only when its **paint( )** method is called by the run-time system. This raises an interesting question: How can the applet itself cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? Remember that one of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the Java run-time system. It cannot create a loop inside **paint( )** that repeatedly scrolls the banner, for example. This would prevent control from passing back to the run-time system. Given this constraint, it may seem that output to your applet's window will be difficult at best. Fortunately, this is not the case. Whenever your applet needs to update the information displayed in its window, it simply calls **repaint( )**.

1. The five methods are **init( )**, **start( )**, **stop( )**, **destroy( )**, and **paint( )**.
2. When **start( )** is called, the applet must be started, or restarted.
3. When **stop( )** is called, the applet must be paused.

The **repaint( )** method is defined by the AWT's **Component** class. It causes the run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**. Thus, for another part of your applet to output to its window, simply store the output and then call **repaint( )**. This causes a call to **paint( )**, which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a **String** variable and then call **repaint( )**. Inside **paint( )**, you will output the string using **drawString( )**.

The simplest version of **repaint( )** is shown here:

```
void repaint( )
```

This version causes the entire window to be repainted.

Another version of **repaint( )** specifies a region that will be repainted:

```
void repaint(int left, int top, int width, int height)
```

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top,* and the width and height of the region are passed in *width* and *height.* These dimensions are specified in pixels. You save time by specifying a region to repaint because window updates are costly in terms of time. If you only need to update a small portion of the window, it is more efficient to repaint only that region.

An example that demonstrates **repaint( )** is found in Project 14-1.

The update( ) Method

There is another method that relates to repainting called **update( )** that your applet may want to override. This method is defined by the **Component** class, and it is called when your applet has requested that a portion of its window be redrawn. The default version of **update( )** simply calls **paint( )**. However, you can override the **update( )** method so that it performs more subtle repainting, but this is an advanced technique that is beyond the scope of this book.

Ask the Expert

Q: **Is it possible for a method other than** paint( ) **or** update( ) **to output to an applet's window?**

A: Yes. To do so, you must obtain a graphics context by calling **getGraphics( )** (defined by **Component**) and then use this context to output to the window. However, for most applications, it is better and easier to route window output through **paint( )** and to call **repaint( )** when the contents of the window change.

Project 14-1 A Simple Banner Applet

Banner.java

To demonstrate **repaint( )**, a simple banner applet is presented. This applet scrolls a message, from right to left, across the applet's window. Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. Banners are popular Web features, and this project shows how to use a Java applet to create one.

Step by Step

1. Create a file called **Banner.java**.
2. Begin creating the banner applet with the following lines.

```
/*
   Project 14-1

   A simple banner applet.

   This applet creates a thread that scrolls
   the message contained in msg right to left
   across the applet's window.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="Banner" width=300 height=50>
</applet>
*/

public class Banner extends Applet implements Runnable {
  String msg = " Java Rules the Web ";
  Thread t;
  boolean stopFlag;

  // Initialize t to null.
  public void init() {
    t = null;
  }
```

Notice that **Banner** extends **Applet**, as expected, but it also implements **Runnable**. This is necessary since the applet will be creating a second thread of execution that will be used to scroll the banner. The message that will be scrolled in the banner is contained in the **String** variable **msg**. A reference to the thread that runs the applet is stored in **t**. The Boolean variable **stopFlag** is used to stop the applet. Inside **init( )**, the thread reference variable **t** is set to **null**.

(continued)

3. Add the **start( )** method shown next.

```
// Start thread
public void start() {
  t = new Thread(this);
  stopFlag = false;
  t.start();
}
```

The run-time system calls **start( )** to start the applet running. Inside **start( )**, a new thread of execution is created and assigned to the **Thread** variable **t**. Then, **stopFlag** is set to **false**. Next, the thread is started by a call to **t.start( )**. Remember that **t.start( )** calls a method defined by **Thread**, which causes **run( )** to begin executing. It does not cause a call to the version of **start( )** defined by **Applet**. These are two separate methods.

4. Add the **run( )** method, as shown here.

```
// Entry point for the thread that runs the banner.
public void run() {
  char ch;

  // Display banner
  for( ; ; ) {
    try {
      repaint();
      Thread.sleep(250);
      ch = msg.charAt(0);
      msg = msg.substring(1, msg.length());
      msg += ch;
      if(stopFlag)
        break;
    } catch(InterruptedException exc) {}
  }
}
```

In **run( )**, the characters in the string contained in **msg** are repeatedly rotated left. Between each rotation, a call to **repaint( )** is made. This eventually causes the **paint( )** method to be called, and the current contents of **msg** are displayed. Between each iteration, **run( )** sleeps for a quarter of a second. The net effect of **run( )** is that the contents of **msg** are scrolled right to left in a constantly moving display. The **stopFlag** variable is checked on each iteration. When it is **true**, the **run( )** method terminates.

5. Add the code for **stop( )** and **paint( )** as shown here.

```
// Pause the banner.
public void stop() {
  stopFlag = true;
```

```
  t = null;
}

// Display the banner.
public void paint(Graphics g) {
  g.drawString(msg, 50, 30);
}
```

If a browser is displaying the applet when a new page is viewed, the **stop( )** method is called, which sets **stopFlag** to **true**, causing **run( )** to terminate. It also sets **t** to **null**. Thus, there is no longer a reference to the **Thread** object, and it can be recycled the next time the garbage collector runs. This is the mechanism used to stop the thread when its page is no longer in view. When the applet is brought back into view, **start( )** is once again called, which starts a new thread to execute the banner.

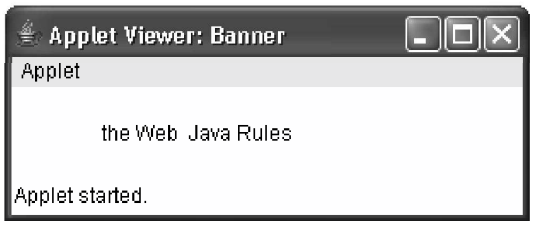
6. The entire banner applet is shown here:

```
/*
   Project 14-1

   A simple banner applet.

   This applet creates a thread that scrolls
   the message contained in msg right to left
   across the applet's window.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="Banner" width=300 height=50>
</applet>
*/

public class Banner extends Applet implements Runnable {
  String msg = " Java Rules the Web ";
  Thread t;
  boolean stopFlag;

  // Initialize t to null.
  public void init() {
    t = null;
  }

  // Start thread
  public void start() {
    t = new Thread(this);
```

(continued)

```
    stopFlag = false;
    t.start();
  }

  // Entry point for the thread that runs the banner.
  public void run() {
    char ch;

    // Display banner
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;
        if(stopFlag)
          break;
      } catch(InterruptedException exc) {}
    }
  }

  // Pause the banner.
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the banner.
  public void paint(Graphics g) {
    g.drawString(msg, 50, 30);
  }
}
```

Sample output is shown here:



CRITICAL SKILL 14.6

Using the Status Window

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **showStatus( )**, which is defined by **Applet**, with the string that you want displayed. The general form of **showStatus( )** is shown here:

void showStatus(String *msg*)

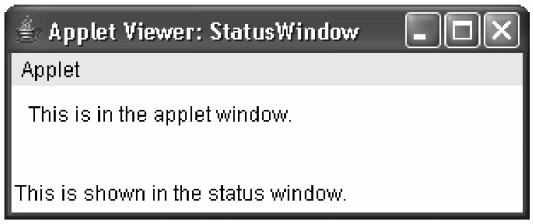
Here, *msg* is the string to be displayed.

The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

The following applet demonstrates **showStatus( )**:

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/

public class StatusWindow extends Applet{
  // Display msg in applet window.
  public void paint(Graphics g) {
    g.drawString("This is in the applet window.", 10, 20);
    showStatus("This is shown in the status window.");
  }
}
```

Sample output from this program is shown here:



CRITICAL SKILL 14.7

Passing Parameters to Applets

You can pass parameters to your applet. To do so, use the PARAM attribute of the APPLET tag, specifying the parameter's name and value. To retrieve a parameter, use the **getParameter( )** method, defined by **Applet**. Its general form is shown here:

String getParameter(String *paramName*)

Here, *paramName* is the name of the parameter. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **boolean** values, you will need to convert their string representations into their internal formats. If the specified parameter cannot be found, **null** is returned. Therefore, be sure to confirm that the value returned by **getParameter( )** is valid. Also, check any parameter that is converted into a numeric value, confirming that a valid conversion took place.

Here is an example that demonstrates passing parameters:

```
// Pass a parameter to an applet.
import java.awt.*;
import java.applet.*;

/*
<applet code="Param" width=300 height=80>
<param name=author value="Herb Schildt">
<param name=purpose value="Demonstrate Parameters">
<param name=version value=2>
</applet>
*/

public class Param extends Applet {
  String author;
  String purpose;
  int ver;

  public void start() {
    String temp;

    author = getParameter("author");
    if(author == null) author = "not found";

    purpose = getParameter("purpose");
    if(purpose == null) purpose = "not found";

    temp = getParameter("version");
```

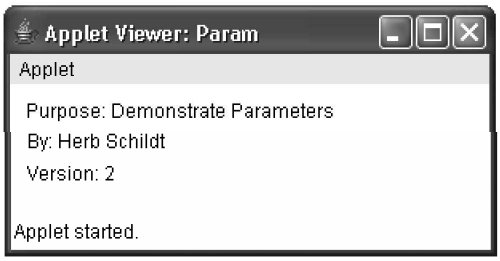
These HTML parameters are passed to the applet.

It is important to check that the parameter exists!

```
    try {
      if(temp != null)
        ver = Integer.parseInt(temp);
      else
        ver = 0;
    } catch(NumberFormatException exc) {
        ver = -1; // error code
    }
  }

  public void paint(Graphics g) {
    g.drawString("Purpose: " + purpose, 10, 20);
    g.drawString("By: " + author, 10, 40);
    g.drawString("Version: " + ver, 10, 60);
  }
}
```

It is also important to make sure that numeric conversions succeed.

Sample output from this program is shown here:



Progress Check

1. How do you cause an applet's **paint( )** method to be called?
2. Where does **showStatus( )** display a string?
3. What method is used to obtain a parameter specified in the APPLET tag?

1. To cause **paint( )** to be called, call **repaint( )**.
2. **showStatus( )** displays output in an applet's status window.
3. To obtain a parameter, call **getParameter( )**.

CRITICAL SKILL 14.8

The Applet Class

As mentioned, all applets are subclasses of the **Applet** class. **Applet** inherits the following superclasses defined by the AWT: **Component**, **Container**, and **Panel**. Thus, an applet has access to the full functionality of the AWT.

In addition to the methods described in the preceding sections, **Applet** contains several others that give you detailed control over the execution of your applet. All of the methods defined by **Applet** are shown in Table 14-1.

Method	Description
void destroy( )	Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.
AccessibleContext getAccessibleContext( )	Returns the accessibility context for the invoking object.
AppletContext getAppletContext( )	Returns the context associated with the applet.
String getAppletInfo( )	Returns a string that describes the applet.
AudioClip getAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*.
AudioClip getAudioClip(URL *url*, String *clipName*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url* and having the name specified by *clipName*.
URL getCodeBase( )	Returns the URL associated with the invoking applet.
URL getDocumentBase( )	Returns the URL of the HTML document that invokes the applet.
Image getImage(URL *url*)	Returns an **Image** object that encapsulates the image found at the location specified by *url*.
Image getImage(URL *url*, String *imageName*)	Returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*.
Locale getLocale( )	Returns a **Locale** object that is used by various locale-sensitive classes and methods.

Table 14-1 The Methods Defined by **Applet**

Method	Description
String getParameter(String *paramName*)	Returns the parameter associated with *paramName*. **null** is returned if the specified parameter is not found.
String[ ] [ ] getParameterInfo( )	Returns a **String** table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose.
void init( )	This method is called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns **true** if the applet has been started. It returns **false** if the applet has been stopped.
static final AudioClip newAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*. This method is similar to **getAudioClip( )** except that it is static and can be executed without the need for an **Applet** object.
void play(URL *url*)	If an audio clip is found at the location specified by *url*, the clip is played.
void play(URL *url*, String *clipName*)	If an audio clip is found at the location specified by *url* with the name specified by *clipName*, the clip is played.
void resize(Dimension *dim*)	Resizes the applet according to the dimensions specified by *dim*. **Dimension** is a class stored inside **java.awt**. It contains two integer fields: **width** and **height**.
void resize(int *width*, int *height*)	Resizes the applet according to the dimensions specified by *width* and *height*.
final void setStub(AppletStub *stubObj*)	Makes *stubObj* the stub for the applet. This method is used by the run-time system and is not usually called by your applet. A *stub* is a small piece of code that provides the linkage between your applet and the browser.
void showStatus(String *str*)	Displays *str* in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.
void start( )	Called by the browser when an applet should start (or resume) execution. It is automatically called after **init( )** when an applet first begins.
void stop( )	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls **start( )**.

Table 14-1 The Methods Defined by **Applet** *(continued)*

Event Handling

Applets are event-driven programs. Thus, event handling is at the core of successful applet programming. Most events to which your applet will respond are generated by the user. These events are passed to your applet in a variety of ways, with the specific method depending upon the actual event. There are several types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the **java.awt.event** package.

Before beginning our discussion of event handling, an important point must be made: The way events are handled by an applet changed significantly between the original version of Java (1.0) and modern versions of Java, beginning with version 1.1. The 1.0 method of event handling is still supported, but it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs, and it is the method described here.

Once again, it must be mentioned that it is not possible to fully discuss Java's event handling mechanism. Event handling is a large topic with many special features and attributes, and a complete discussion is well beyond the scope of this book. However, the overview presented here will help you get started.

CRITICAL SKILL 14.9

The Delegation Event Model

The modern approach to handling events is based on the *delegation event model*. The delegation event model defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification.

Events

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface, such as pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event Sources

An event source is an object that generates an event. A source must register listeners in order for the listener to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines methods that receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. The class **AWTEvent**, defined within the **java.awt** package, is a subclass of

EventObject. It is the superclass (either directly or indirectly) for all AWT-based events used by the delegation event model.

The package **java.awt.event** defines several types of events that are generated by various user interface elements. Table 14-2 enumerates the most commonly used ones and provides a brief description of when they are generated.

Event Listener Interfaces

Event listeners receive event notifications. Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table 14-3 lists commonly used listener interfaces and provides a brief description of the methods they define.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged or moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 14-2 The Main Event Classes in **java.awt.event**

Interface	Description
ActionListener	Defines one method to receive action events. Action events are generated by such things as push buttons and menus.
AdjustmentListener	Defines one method to receive adjustment events, such as those produced by a scroll bar.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes. An item event is generated by a check box, for example.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
TextListener	Defines one method to recognize when a text value changes.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 14-3 Common Event Listener Interfaces

Progress Check

1. Briefly explain the significance of **EventObject** and **AWTEvent**.
2. What is an event source? What is an event listener?
3. Listeners must be registered with a source in order to receive event notifications. True or False?

1. **EventObject** is a superclass of all events. **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.
2. An event source generates events. An event listener receives event notifications.
3. True.

CRITICAL SKILL 14.10

Using the Delegation Event Model

Now that you have had an overview of the delegation event model and its various components, it is time to see it in practice. Applet programming using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

To see how the delegation model works in practice, we will look at an example that handles one of the most commonly used event generators: the mouse.

Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. The **MouseListener** interface defines five methods. If a mouse button is clicked, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when a mouse button is pressed and released, respectively. The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

The **MouseMotionListener** interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)

void mouseMoved(MouseEvent *me*)

The **MouseEvent** object passed in *me* describes the event. **MouseEvent** defines a number of methods that you can use to get information about what happened. Probably the most commonly used methods in **MouseEvent** are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse when the event occurred. Their forms are shown here:

int getX( )

int getY( )

The next example will use these methods to display the current location of the mouse.

A Simple Mouse Event Applet

The following applet demonstrates handling mouse events. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upper-left corner of the applet display area.

As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint( )** to display output at the point of these occurrences.

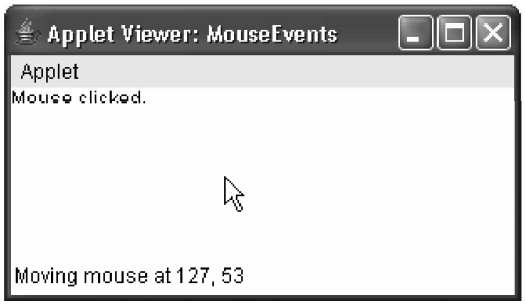
```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MouseEvents" width=300 height=100>
  </applet>
*/

public class MouseEvents extends Applet
  implements MouseListener, MouseMotionListener {

  String msg = "";
  int mouseX = 0, mouseY = 0; // coordinates of mouse

  public void init() {
     addMouseListener(this);
     addMouseMotionListener(this); <-- Register this class as a listener for mouse events.
  }
```

```
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse clicked.";
  repaint();
}

// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse entered.";
  repaint();
}

// Handle mouse exited.
public void mouseExited(MouseEvent me) {
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse exited.";
  repaint();
}

// Handle button pressed.
public void mousePressed(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Down";
  repaint();
}

// Handle button released.
public void mouseReleased(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Up";
  repaint();
}

// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
  // save coordinates
  mouseX = me.getX();
  mouseY = me.getY();
```

This, and the other event handlers, respond to mouse events.

```
    msg = "*";
    showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
    repaint();
  }

  // Handle mouse moved.
  public void mouseMoved(MouseEvent me) {
    // show status
    showStatus("Moving mouse at " + me.getX() + ", " +
               me.getY());
  }

  // Display msg in applet window at current X,Y location.
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
  }
}
```

Sample output from this program is shown here:



Let's look closely at this example. The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events. This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets.

Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which are members of **Component**. They are shown here:

void addMouseListener(MouseListener *ml*)

void addMouseMotionListener(MouseMotionListener *mml*)

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both.

The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

CRITICAL SKILL 14.11

More Java Keywords

Before concluding this book, a few more Java keywords need to be briefly discussed:

- **transient**
- **volatile**
- **instanceof**
- **native**
- **strictfp**
- **assert**

These keywords are used in programs more advanced than those found in this book. However, an overview of each is presented so that you will know their purpose.

The transient and volatile Modifiers

The **transient** and **volatile** keywords are type modifiers that handle somewhat specialized situations. When an instance variable is declared as **transient**, then its value need not persist when an object is stored. Thus, a **transient** field is one that does not affect the state of an object.

The **volatile** modifier was mentioned briefly in Module 11 but warrants a closer look. Modifying a variable with **volatile** tells the compiler that the variable can be changed unexpectedly by other parts of your program. As you saw in Module 11, one of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads will share the same instance variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable. The real (or *master*) copy of the variable is updated at various times, such as when a **synchronized** method is entered. While this approach works fine, there may be times when it is inappropriate. In some cases, all that really matters is that the master copy of a variable always reflects the current state. To ensure this, simply specify the variable as **volatile**. Doing so tells the compiler that it must always keep any private copies up to date with the master copy and vice versa. Also, accesses to the

master variable must be executed in the precise order in which they are executed on any private copy.

instanceof

Sometimes it is useful to know the type of an object during run time. For example, you might have one thread of execution that generates various types of objects and another thread that processes these objects. In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object's type at run time is important involves casting. In Java, an invalid cast causes a run-time error. Many invalid casts can be caught at compile time. However, casts involving class hierarchies can produce invalid casts that can only be detected at run time. Because a superclass reference can refer to subclass objects, it is not always possible to know at compile time whether or not a cast involving a superclass reference is valid. The **instanceof** keyword addresses these types of situations.

The **instanceof** operator has this general form:

object instanceof *type*

Here, *object* is an instance of a class, and *type* is a class or interface type. If *object* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**. Otherwise, its result is **false**. Thus, **instanceof** is the means by which your program can obtain run-time type information about an object.

strictfp

One of the more esoteric keywords is **strictfp**. With the creation of Java 2, the floating-point computation model was relaxed slightly to make certain floating-point computations faster for certain processors, such as the Pentium. Specifically, the new model does not require the truncation of certain intermediate values that occur during a computation. By modifying a class or a method with **strictfp**, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java. The truncation affects only the exponent of certain operations. When a class is modified by **strictfp**, all of the methods in the class are also **strictfp** automatically.

assert

The **assert** keyword is used during program development to create an *assertion*, which is a condition that is expected to be true during the execution of the program. For example, you

might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an **assert** statement. At run time, if the condition actually is true, no other action takes place. However, if the condition is false, then an **AssertionError** is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.

The **assert** keyword has two forms. The first is shown here:

assert *condition*;

Here, *condition* is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default **AssertionError** object is thrown. For example,

```
assert n > 0;
```

If **n** is less than or equal to zero, then an **AssertionError** is thrown. Otherwise, no action takes place.

The second form of **assert** is shown here:

assert *condition* : *expr*;

In this version, *expr* is a value that is passed to the **AssertionError** constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for *expr*, but any non-**void** expression is allowed as long as it defines a reasonable string conversion.

To enable assertion checking at run time, you must specify the **-ea** option. For example, to enable assertions for **Sample**, execute it using this line:

```
java -ea Sample
```

Assertions are quite useful during development because they streamline the type of error checking that is common during testing. But be careful—you must not rely on an assertion to perform any action actually required by the program. The reason is that normally, released code will be run with assertions disabled and the expression in an assertion will not be evaluated.

Native Methods

Although rare, there may occasionally be times when you will want to call a subroutine that is written in a language other than Java. Typically, such a subroutine will exist as executable code for the CPU and environment in which you are working—that is, native code. For example, you may wish to call a native code subroutine in order to achieve faster execution time. Or you may

want to use a specialized, third-party library, such as a statistical package. However, since Java programs are compiled to bytecode, which is then interpreted (or compiled on the fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Fortunately, this conclusion is false. Java provides the **native** keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the **native** modifier, but do not define any body for the method. For example:

```
public native int meth() ;
```

Once you have declared a native method, you must write the native method and follow a rather complex series of steps in order to link it with your Java code.

What Next?

Congratulations! If you have read and worked through the preceding 14 modules, then you can call yourself a Java programmer. Of course, there are still many, many things to learn about Java, its libraries, and its subsystems, but you now have a solid foundation upon which you can build your knowledge and expertise.

Here are a few of the topics that you will want to learn more about:

- The Abstract Window Toolkit (AWT), including its various user interface elements, such as push buttons, menus, lists, and scroll bars.
- Layout managers, which control how elements are displayed in an applet.
- Handling text and graphics output in a window.
- The event handling subsystem. Although introduced here, there is substantially more to it.
- Java's networking classes.
- Java's utility classes, especially its Collections Framework, which simplifies a number of common programming tasks.
- The Concurrent API, which offers detailed control over high-performance multithreaded applications.
- Swing, which is an alternative to the AWT.
- Java Beans, which supports the creation of software components in Java.
- Creating native methods.