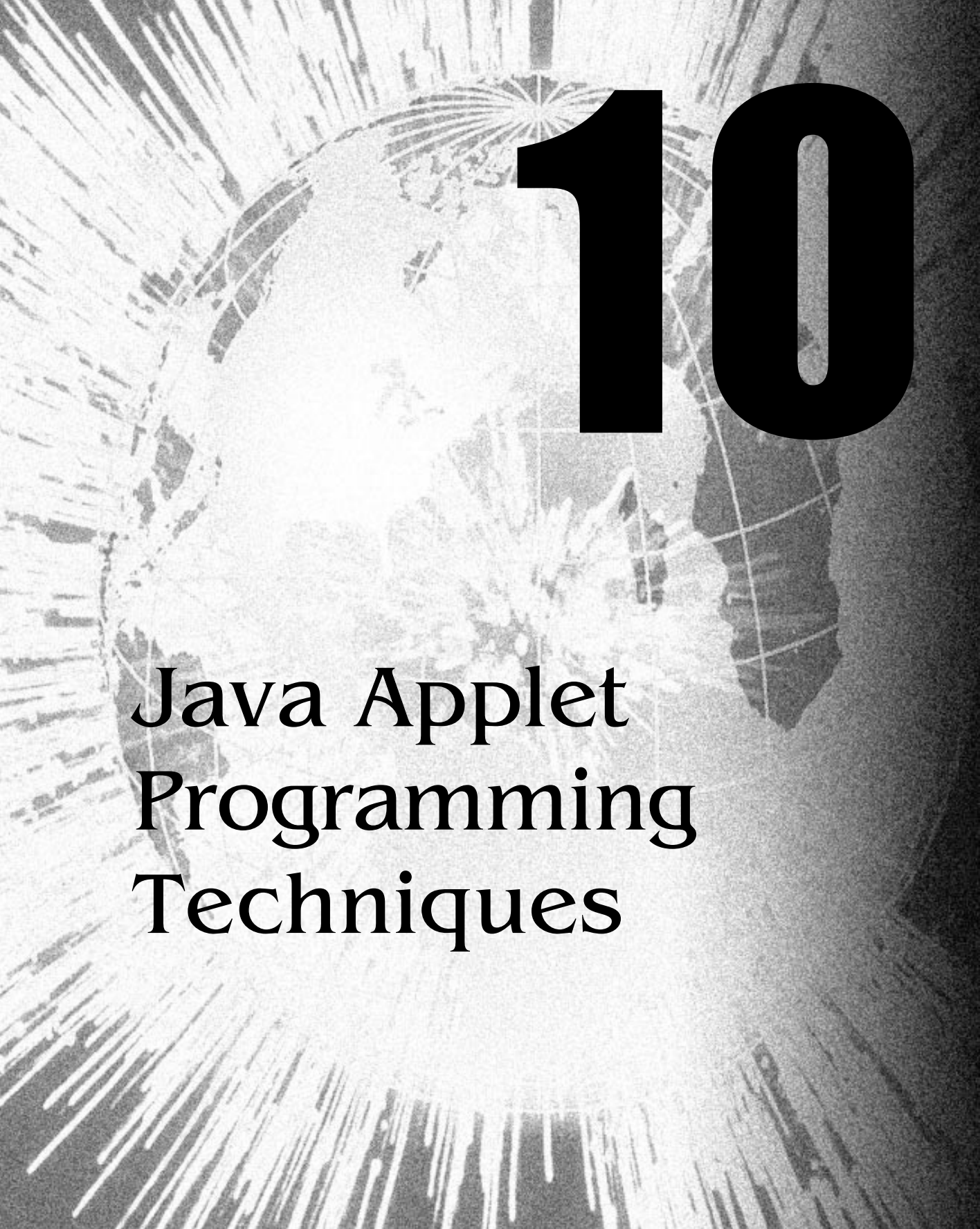
JAVA

PROGRAMMING LANGUAGE HANDBOOK

Anthony Potts

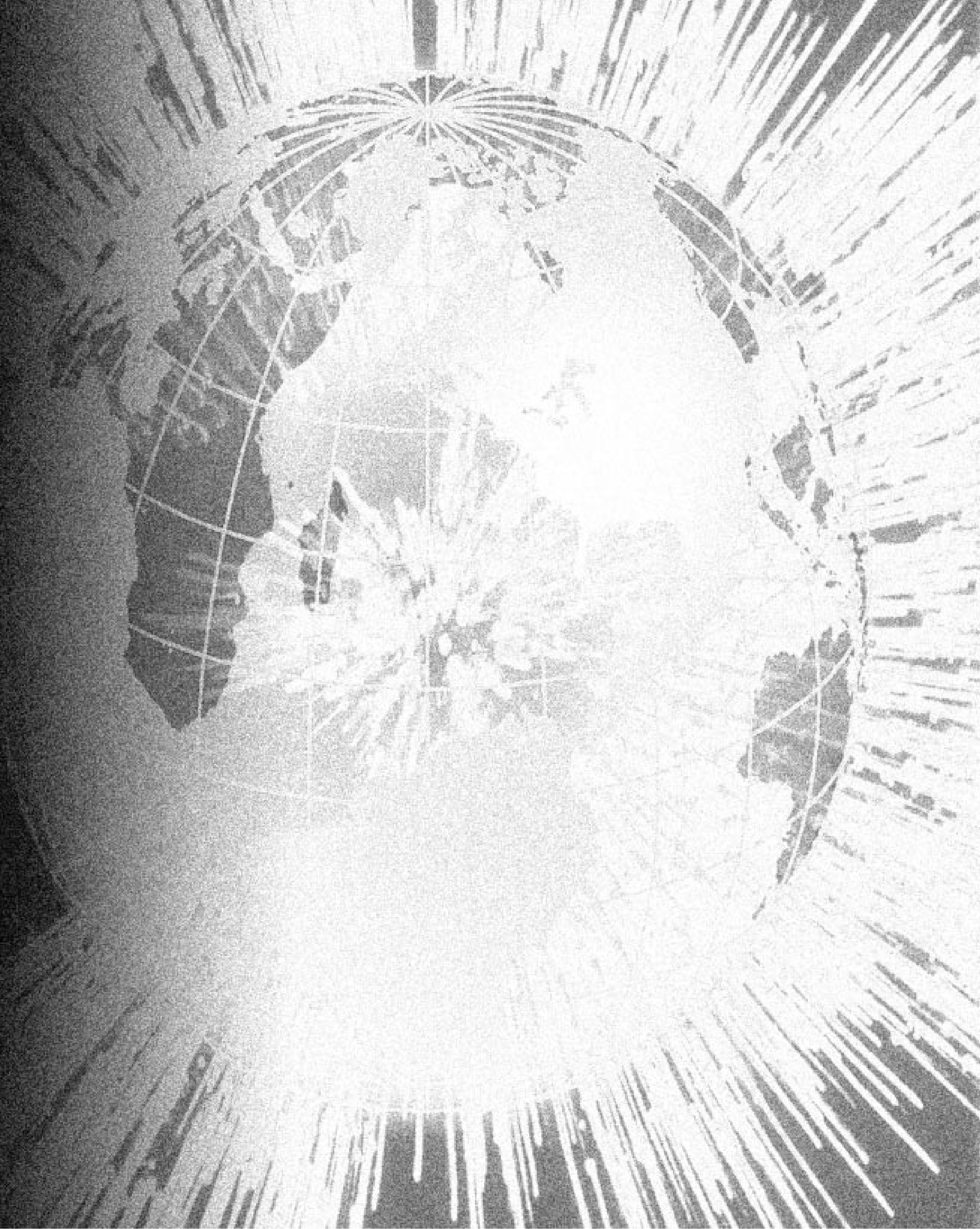
David H. Friedel, Jr.

CORIOLIS GROUP BOOKS



10

Java Applet Programming Techniques



10 Java Applet Programming Techniques

Once you master the basics of using the Java language, you'll want to learn as much as you can about writing powerful applets.

The Java language offers a unique option—to be able to create programs that run as a "stand-alone" applications or as applets that are dependent on a controlling program such as a Web browser. The big difference between applications and applets is that applications contain enough code to work on their own and applets need a controlling program to tell the applet when to do what.

By itself, an applet has no means of starting execution because it does not have a **main**() method. In an application, the **main**() method is the place where execution starts. Any classes that are not accessed directly or indirectly through the **main**() method of an application are ignored.

If you have programmed in a visual environment before, the concept of an applet should be easy to understand. You can think of an applet as you would a type of custom control. When a custom control is used, you don't have to create code to make it go; that is handled for you. All you have to do is respond to events. With applets, you do not have to create the code that makes it go; you only need to write code to respond to events that the parent program calls—usually the browser.

Applet Basics

Let's look closely at some of the key areas you need to be aware of when creating applets. To start, you need to subclass the **Applet** class. By doing this, you inherit quite a bit of applet functionality that is built-in to the **Applet** class.

This listing shows the hierarchy of the Applet class and Table 10.1 provides a detailed look at the components of the **Applet** class that are inherited when you implement it.

Hiererachy of the Applet Class

```
java.lang.Object
     java.awt.Component
          java.awt.Container
               java.awt.Panel
                    java.applet.Applet
```

Table 10.1 Methods Available in the Applet Class

Method	Description
destroy()	Cleans up whatever resources are being held. If the applet is active, it is first stopped.
getAppletContext()	Returns a handle to the applet context. The applet context is the parent object—either a browser or applet viewer. By knowing this handle, you can control the environment and perform operations like telling a browser to download a file or jump to another Web page.
getAppletInfo()	Returns a string containing information about the author, version, and copyright of the applet.
getAudioClip(URL)	Returns the data of an audio clip that is located at the given URL. The sound is not played until the **play()** method is called.
getAudioClip(URL, String)	Returns the data of an audio clip that is located at the given location relative to the document's URL. The sound is not played until the **play()** method is called.
getCodeBase()	Returns the URL of the applet itself.
getDocumentBase ()	Gets the URL of the document that the applet is embedded in. If the applet stays active as the browser goes from page to page, this method will still return the URL of the original document the applet was called from.
getImage(URL)	Gets an image at the given URL. This method always returns an image object immediately even if the image does not exist. The actual image details are loaded when the image is first needed and not at the time it is loaded. If no image exists, an exception is thrown.
getImage(URL, String)	Gets an image at a URL relative to the document's URL. This method always returns an image object immediately even if the image does not exist. The actual image details are loaded when the image is first needed and not at the time it is loaded. If no image exists, an exception is thrown.
getParameter(String)	Returns a parameter that matches the value of the argument string.

continued

Table 10.1 Methods Available in the Applet Class (Continued)

Method	Description
getParameterInfo()	Returns an array of strings describing the parameters that are understood by this applet. The array consists of sets of three strings: name, type, and description. Often, the description string will be empty.
init()	Initializes the applet. You never need to call this method directly; it is called automatically by the system once the applet is created. The **init()** method is empty so you need to override it if you need anything initialized at the time your applet is loaded.
isActive()	Returns true if the applet is active. An applet is marked active just before the **start()** method is called.
play(URL)	This method plays an audio clip that can be found at the given URL. Nothing happens if the audio clip cannot be found.
play(URL, String)	This method plays an audio clip that resides at a location relative to the current URL. Nothing happens if the audio clip is not found.
resize(int, int)	Requests that an applet be resized. The first integer is height and the second is width. This method overrides the **resize()** method of the **Component** class that is part of the **Applet** class's hierarchy.
showStatus(String)	Shows a status message in the Applet's context. This method allows you to display a message in the applet context's status bar, usually a browser. This is very useful for displaying URL's when an action the user is about to do will result in a jump to a new Web page.
start()	Starts the applet. You never need to call this method directly; it is called when the applet's document is visited. Once again, this method is empty and you need to override it to make it useful. This is usually where you would put the guts of your code. However, be aware that this method is called *every* time the page that embeds the applet is called. So make sure that the applet is being destroyed if necessary.
stop()	This method is called when the browser leaves the page the applet is embedded in. It is up to you to take this opportunity to use the **destroy()** method to terminate your applet. There may be times, however, when you do not want to destroy it here and instead wait until a "Quit" button is pressed, or until the browser itself closes (then everything is dumped rather ungraciously). **stop()** is guaranteed to be called before the **destroy()** method is called. You never need to call this method directly because the browser will call it for you.

If you return to Chapter 2 and walk through the ticker tape applet we presented, you should get a good overview of the order in which key methods are called and why. As the listing illustrates, the **Applet** class is a descendant of the

Container class; thus, it can hold other objects. You do not need to create a panel first to place objects on because the **Applet** class extends the **Panel** class. Finally, because the **Container** class is derived from the **Component** class, we have the ability to respond to events, grab and display images, and display text among many other things. Table 10.2 presents some of the key methods you have access to because of all the classes that have been extended to get to the **Applet** class.

Table 10.2 Key Methods That the Applet Class Can Use

Derived from the Component class:

Method	Description
getBackground()	Gets the background color. If the component does not have a background color, the background color of its parent is returned.
getFont()	Gets the font of the component. If the component does not have a font, the font of its parent is returned.
getFontMetrics(Font)	Gets the font metrics for this component. It will return null if the component is currently not on the screen. Font metrics tell you things like the height and width of a string using the given font on the current component.
getForeground()	Gets the foreground color. If the component does not have a foreground color, the foreground color of its parent is returned.
getGraphics()	Gets a Graphics context for this component. This method will return null if the component is currently not visible on the screen.
handleEvent(Event)	Handles all events. Returns true if the event is handled and should not be passed to the parent of this component. The default event handler calls some helper methods to make life easier on the programmer.
hide()	Hides the component.
inside(int, int)	Checks if a specified x,y location is inside this component.
locate(int, int)	Returns the component or subcomponent that contains the x,y location.
location()	Returns the current location of this component. The location will be in the parent's coordinate space. The return value is a **point** object which is simply an x and y integer value.
move(int, int)	Moves the component to a new location. The integers are the x and y coordinates and are in the parent's coordinate space.
repaint()	Repaints the component. This will result in a call to update as soon as possible. The screen will also be cleared resulting in a brief flicker.
repaint(long)	Repaints the component. However, the extra argument is a **long** value that instructs Java that it must perform the update within that value in milliseconds.

continued

Table 10.2 Key Methods That the Applet Class Can Use (Continued)

Method	Description
reshape(int, int, int, int)	Reshapes the component to the specified bounding box. The first two integers represent the new x an y coordinates the component should be moved to, and the second set of integers represent the new width and height.
resize(int, int)	Resizes the component to the specified width and height.
setBackground(Color)	Sets the background color.
setFont(Font)	Sets the font of the component.
setForeground(Color)	Sets the foreground color.
show()	Shows the component.
size()	Returns the current size of the component. The return value is a dimension object that has two integer values representing the width and height.
update(graphics)	Updates the component. This method is called in response to a call to **repaint()**. If you override this method and call it instead of the **paint()** method, the screen will not be cleared first.
Derived from the Container class:	
add(Component)	Adds the specified component to this container.
countComponents()	Returns the number of components in this panel.
getComponents()	Gets all the components in this container. The return value is actually an array of **Component** objects.
getLayout()	Returns the layout manager for the container.
remove(Component)	Removes the specified component from the container.
removeAll()	Removes all the components from the container. It is dangerous to use if you are not careful.
setLayout(LayoutManager)	Sets the layout manager for the container.

Applet Drawbacks

Applets can really eat up system resources. If you do not use threads and you create loops in an applet, you may run into serious performance problems with your browser. The browser can get so busy working with the applet that it does not have time to respond to Web page events such as refreshes, scrolls, and mouse clicks.

When some developers first heard about applets and their ability to run on many types of machines, there first response was, "That's dangerous!" Many were concerned that applets would be used for mischievous causes. To prevent

applets from causing problems on machines, there are several built-in security measures you can take advantage of.

Limited File Access

Applets cannot read from or write to files on an end user's hard drive. They can only read files that reside on the machine the applet was called from. Eventually, a user will be able to set up specific directories that an applet can have access to, but that functionality is not very robust yet and may not be implemented on all browsers, so don't count on it.

Native Methods

The other option or loop-hole (depending on how you look at it) is the use of *native methods*. You can create methods in C++ that can be called directly from Java. This is a very powerful option, especially if you are creating platform-specific programs that need the extra speed that you can get from natively compiled code. However, it can also be a potential gateway for mischievous programs. This feature may or may not be disabled, depending on the browser, so be cautious of how you use it.

File Execution

Java applets are not allowed to execute programs on a user's system. So, you can't just run the **Format** program and wipe a hard drive.

Network Communication

Applets are only allowed to communicate with the server from which they were downloaded. This is another one of the security features that may or not be in effect depending on the browser. So, once again, do not program for it. This is actually one security option we would like to see go away or at least be able to have the user override it. The ability to talk with multiple servers could be incredibly powerful if implemented well. Just think of a large company with servers all over the world. You could create a little applet that could converse with them all and gather information for the end users.

A Disclaimer

Just because Java provides a few security features does not mean that it is completely secure. Java is a language that is still very much in its infancy and someone, somewhere will find a way to hack the system. However, since Java was

produced to be an Internet friendly language (one of the first), it is much more secure than other languages. The problem is that it is also getting much more attention than all the others combined. Attention from users, programmers, and *hackers*!

Let's Play

Now that you have seen all the methods that you can use and learned a little about applet security, let's create an applet that uses some of these features. We won't explain every line of code as we did for the ticker tape applet in Chapter 2, but we will show you a few cool things you can do in your applets.

The applet we'll create is a simple navigation applet that will offer the user several buttons with URLs as labels. When the user clicks on a button, the browser will be instructed to go to a particular site. We have also included some sound support just for the fun of it. Lets see the code first:

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

// testNav Class
public class testNav extends Applet {
   AudioClip startClip;
   AudioClip linkClip;
   AudioClip stopClip;

   public testNav() {
      setLayout(new GridLayout(4, 1));
      add(new Button("http://www.coriolis.com"));
      add(new Button("http://www.javasoft.com"));
      add(new Button("http://www.gamelan.com"));
      add(new Button("http://www.microsoft.com"));
   }

   public boolean action(Event evt, Object arg) {
      if (evt.target instanceof Button) {
         linkClip.play();
         fetchLink((String)arg);
         return true;
      }
      return super.handleEvent(evt);
   }
```

```
      void fetchLink(String s) {
      URL tempURL = null;
      try { tempURL  = new URL(s); }
      catch(MalformedURLException e) {
         showStatus("Malformed URL Exception has been thrown!");
      }
      getAppletContext().showDocument(tempURL);
   }

   public void init(){
      startClip = getAudioClip(getCodeBase(),  "start.au");
      linkClip = getAudioClip(getCodeBase(), "link.au");
      stopClip = getAudioClip(getCodeBase(), "stop.au");
   }

   public void start(){
      testNav TN = new testNav();
      startClip.play();
   }

   public void stop(){
      stopClip.play();
   }

} // End testNav
```

Figure 10.1 shows the testNav applet running in the Netscape browser.

Interacting with the Browser

Quite often, you may want your applet to interact with the host browser. That interaction will usually come in the form of asking the browser to go to another Web site, or changing the text displayed in the status bar at the bottom of the browser. Let's look at how to switch Web pages now, then we'll show you how to control the status bar.

For our little demo program, we need the browser to change pages whenever a button is pushed. To accomplish this, we need to use an event handling method. The **action**() method is the best place to do this, so let's look at that method in more detail:

```
public boolean action(Event evt, Object arg) {
      if (evt.target instanceof Button) {
```

```
            linkClip.play();
            fetchLink((String)arg);
            return true;
        }
        return super.handleEvent(evt);
    }
```

This method handles mouse and keyboard actions for us. The first thing it does is check to see if the **target** of the action is a **Button** or not. If it is, we play a sound file and call the **fetchLink**() method that we created to actually go to other sites. We will cover the sound stuff in a minute. Right now, let's look at the **fetchLink**() method and see how we instruct the browser to grab other pages:

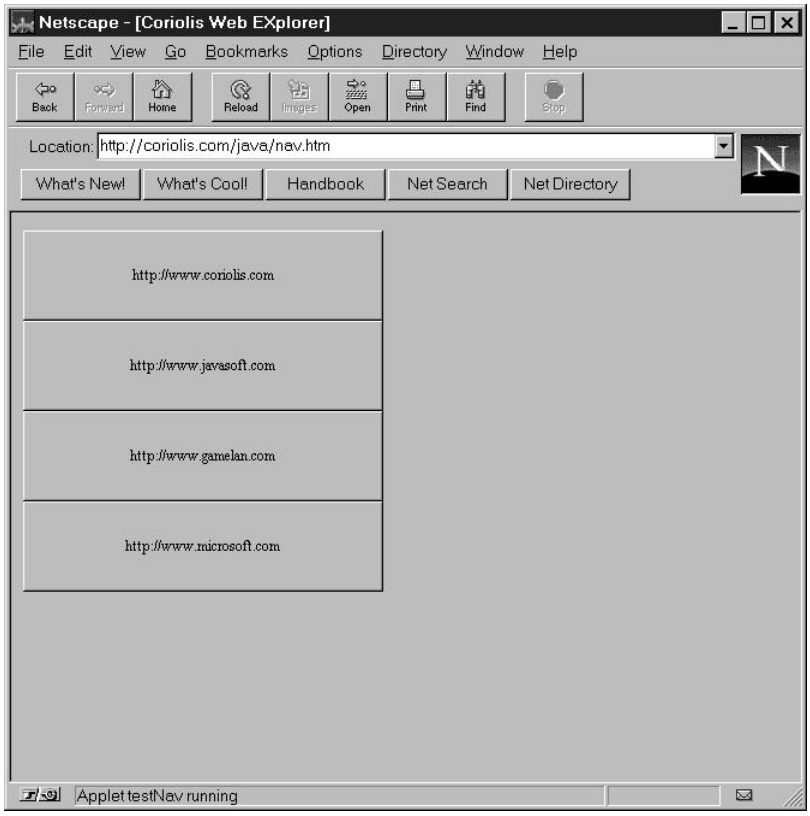


Figure 10.1
Running the testNav applet.

```
void fetchLink(String s) {
      URL tempURL = null;
      try { tempURL  = new URL(s); }
      catch(MalformedURLException e) {
         showStatus("Malformed URL Exception has been thrown!");
      }
      getAppletContext().showDocument(tempURL);
   }
```

This method accepts a string representation of a URL, changes it into a URL object, then calls the **showDocument**() method that really does the work. We are forced to use a **try...catch** operation when we are creating a URL because it throws exceptions. In particular, it throws the **MalformedURLException**. Basically, if the URL string you are trying to turn into a URL object is poorly constructed, you will get an error. For example, if you leave off the "http://" part, you will get this error.

Once the URL is properly created, we call the **showDocument**() method that actually belongs to the browser. This is not an applet method. You can figure this out because we are calling the **getAppletContext**() method at the beginning of the line. This method returns the object representation of the browser which has its own methods, variables, and so on.

Changing the Status Bar

If you look at the **action**() method again, you will notice that we make an interesting method call whenever there is an error. Here is that line:

```
showStatus("Malformed URL Exception has been thrown!");
```

You can also code this operation like this:

```
getAppletContext().showStatus("Malformed URL Exception has been thrown!");
```

To some people, this is easier to read because it becomes immediately apparent which object is accepting the **showStatus**() method.

Changing this text at key times is a great way to interact with the user because the status bar is a consistent object across many applications so they expect it to be there and they expect useful information from it. For a little test, try and make the status bar display the link for any button that the mouse pointer is moving over.

Playing Sounds

For loading and playing sounds from within an applet we have two options. First, we can use the **play**() method of the applet that loads and plays the given sound right away. Second, we can load an applet into an **AudioClip** object using the **getAudioClip**() method and play the sound whenever we want. It's up to you to decide if the sounds should be loaded before they are played, or loaded and played at the same time.

To use the **play**() method, you invoke the method, sending the URL of the sound file you want as an argument. Or, you can split the URL into two pieces. The first piece would be a URL representing the code base, and the second argument would be the file name and directory relative to the code base. Here are the declarations for these methods:

```
play(URL);  // This is the full URL of the sound file you want to play
play(URL, String);  // This is the call that uses a base URL and a string
                    // representing the file name.
```

The other option we have for playing sounds is to load them into an object first. To do this we will create an **AudioClip** object and use the **getAudioClip**() method to load the sound file into the audio object. Once the sound file is loaded, we call the **play**() method of the **AudioClip** object to hear the sound. Here are the declarations and calls to handle sounds in this manner:

```
getAudioClip(URL); // This requires a fully-qualified URL that points to a
                   // sound file
getAudioClip(URL, String); // This is the call that uses a base URL and a
                           // string representing the file name.
```

To declare an **AudioClip** object, just follow this code:

```
AudioClip myClip;
```

Then, to load in the image do this:

```
myClip = getAudioClip(soundURL);
```

Finally, here's the call needed to play the file:

```
myClip.play();
```

You can also stop or loop the sound clip with these methods:

```
myClip.stop();
myClip.loop();
```

If a sound file being requested cannot be found, the **AudioClip** object will be set to null. No exception will be raised, but if you then try to play, stop, or loop the file, an exception will be thrown.

Displaying Images

One other key area we need to cover is the quick and painless use of images within applets. Images are just as easy to download as sounds. Here is a little sample applet that downloads an image and blasts it onto the screen:

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

public class testImg extends Applet {
   Image testImage;

   public void paint(Graphics g) {
      g.drawImage(testImage, 0, 0, this);
   }

   public void init() {
      testImage = getImage(getDocumentBase(), "sample.gif");
   }
}
```

This is an extremely simple program but it illustrates how easy downloading images is. The syntax for downloading images is almost identical to what we used for downloading sounds.

The syntax for declaring an image object is :

```
Image myImage;
```

And the syntax for the **getImage**() method is:

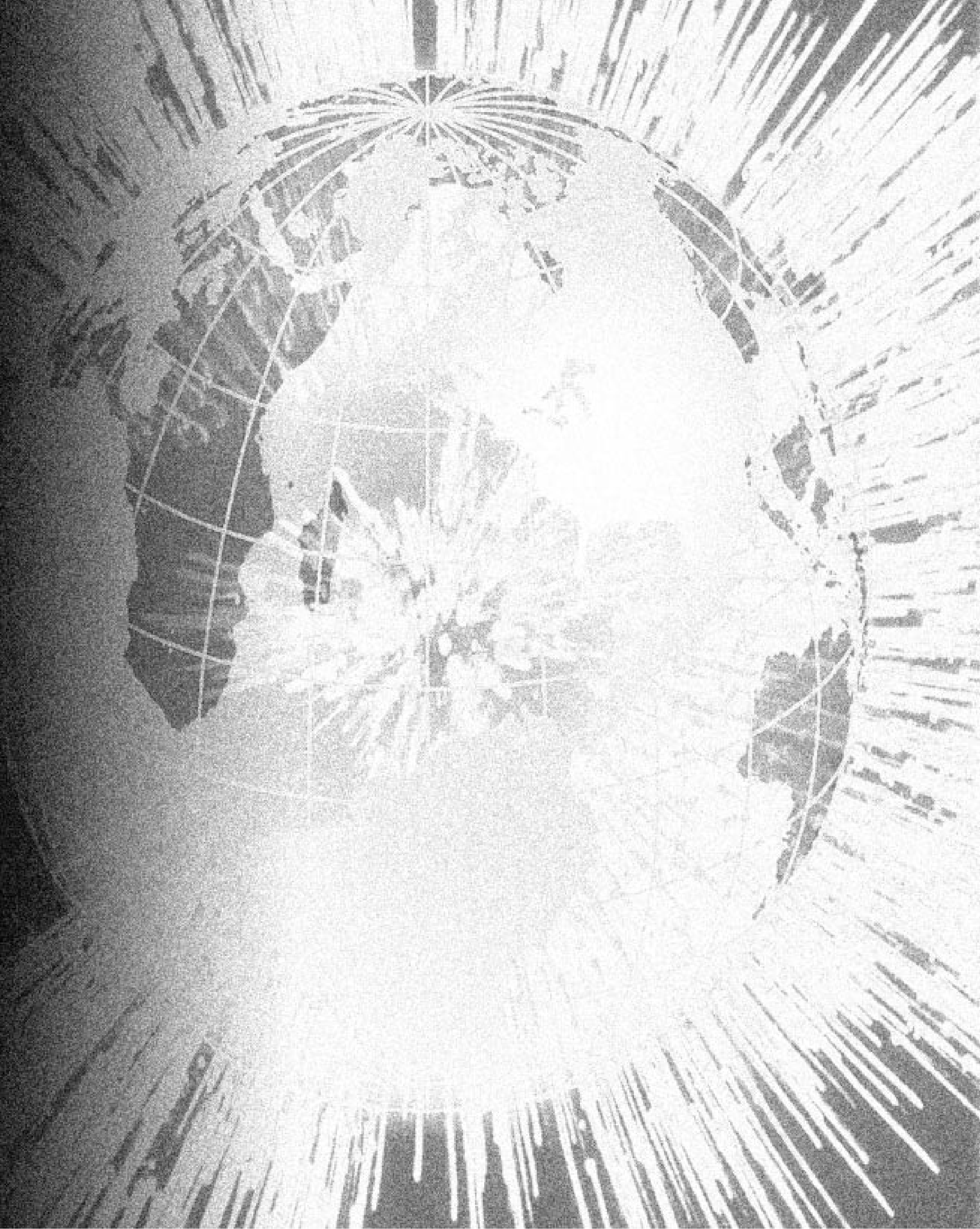
```
getImage(URL); // Downloads the image that resides at the given
               // fully-qualified URL
```

```
getImage(URL, String); // The URL is the code or document base,
                       // and the string is the directory and file name
                       // for the image relative to the code base
```

These image methods will support whatever format the browser supports.

11

Event Handling



11 Event Handling

Whether you use Java to write applications or applets, you'll need to master the art of handling events.

Every time you perform an action while running a Java application, such as clicking a mouse button or dragging a window, an event occurs. But, of course, events can also be triggered by internal actions that occur within a program. Many of the common events that occur in Java programs are handled by classes in the AWT package we discussed in Chapter 9. However, we decided to give events a chapter of their own because of their importance. This way, we can focus on events without being sidetracked by GUI creation issues.

We'll start by introducing the basics of how events are handled in Java. Then we'll present the **Events** class, which is used to derive objects for handling events in Java programs. As you'll see, this class defines a number of instance variables and methods to help process the different types of events that can occur in a Java program. After we cover the essentials of the **Events** class, we'll dig in and look at some of the specific methods that are triggered when events occur. As we explore different types of events, we'll present a number of programming examples to illustrate different techniques available for processing events.

Introducing Events

In the ticker tape applet we created in Chapter 2, the program scrolled a line of text across the applet space. If you click the mouse button on top of the applet while it is running, the scrolling text will stop and then start when the mouse is clicked again. These mouse clicks cause events to occur. In our applet, the event

was caused by pressing down the mouse button. Here is the portion of code responsible for halting the scrolling text:

```
// Handle mouse clicks
   public boolean handleEvent(Event evt) {
      if (evt.id == Event.MOUSE_DOWN) {
         if (suspended) {
            ttapeThread.resume();
         } else {
            ttapeThread.suspend();
         }
      suspended = !suspended;
      }
      return true;
   }
```

The key to the inner workings of this error handler method (**handleEvent**()) is the argument **evt**. It is declared as an object of the **Event** class, a special class that Java provides for processing events. In our code, we simply check the **id** instance variable to make sure a **MOUSE_DOWN** event has occurred. If so, we either resume or suspend the applet.

Event Types

Events in Java can be split into three main groups: *mouse*, *keyboard*, and *system events*. All of these events can be handled very similarly. Java events are actually objects derived from their own classes, as we saw in the applet example we just discussed. This method of handling events makes perfect sense when you realize the power you gain from being able to manipulate an event as an object. In other programming languages, events only trigger certain methods, which limits you to receiving very little information about the current state of the system. You also cannot pass the event on to another handler, which you can do with Java.

The Event Class

Let's take a close look at the **Event** class so we can use it throughout this chapter. This class has many variables and methods that can be used for finding out information about an event that has occurred, such as where and when the event has happened, and who it has happened to. Many of the variables give us status information, such as if the Shift or Page Up key was pressed when the event has occurred.

Table 11.1 presents the variables defined in the **Event** class and Table 11.2 presents the methods. Later in this chapter you'll learn more about how to apply them. The variables that are listed in all capital letters represent static values that

Table 11.1 Variables Defined in the Events Class

Variable	Description
SHIFT_MASK	The shift modifier constant. This is an integer that indicates if the Shift key was down when the event occurred. This variable is used to process keyboard events.
CTRL_MASK	The control modifier constant. This is an integer that indicates if the Ctrl key was down when the event occurred. This variable is used to process keyboard events.
ALT_MASK	The alt modifier constant. This is an integer that indicates if the Alt key was down when the event occurred. This variable is used to process keyboard events.
HOME	Represents the Home key.
END	Represents the End key.
PGUP	Represents the Page Up key.
PGDN	Represents the Page Down key.
UP	Represents the Up arrow key.
DOWN	Represents the Down arrow key.
LEFT	Represents the left arrow key.
RIGHT	Represents the right arrow key.
F1 ... F12	Represents one of the function keys.
ESC	Represents the escape key.
WINDOW_DESTROY	Represents the event that occurs when a user tries to close a frame or window.
WINDOW_EXPOSE	Represents the event that occurs when part of your application has been covered by another application and the second app is removed.
WINDOW_ICONIFY	Represents the event that occurs when a window is minimized.
WINDOW_DEICONIFY	Represents the event that occurs when a window is restored from a minimized state.
WINDOW_MOVED	Represents the event that occurs when the window is moved.
KEY_PRESS	Represents the event that occurs when any key on the keyboard is pressed down.
KEY_RELEASE	Represents the event that occurs when any key on the keyboard is released.
MOUSE_DOWN	Represents the event that occurs when a mouse button is pressed down.
MOUSE_UP	Represents the event that occurs when a mouse button is released.
MOUSE_MOVE	Represents the event that occurs when a mouse button is moved across a part of the application or applet.

continued

Table 11.1 Variables Defined in the Events Class (continued)

Variable	Description
MOUSE_ENTER	Represents the event that occurs when a mouse enters a component.
MOUSE_EXIT	Represents the event that occurs when a mouse exits a component.
MOUSE_DRAG	Represents the event that occurs when the mouse button is down and the mouse is moved.
LIST_SELECT	Represents the event that occurs when an option is selected from within a list object.
LIST_DESELECT	Represents the event that occurs when an option is de-selected from within a list object.
GOT_FOCUS	Represents the event that occurs when a component gains the focus.
LOST_FOCUS	Represents the event that occurs when a component loses the focus.
Target	Holds an object that was the "target" of an event.
When	Indicates the precise time when an event occurred.
Id	Indicates the type of event.
X	The x coordinate of the event.
Y	The y coordinate of the event.
Key	The key that was pressed in a keyboard event.
Arg	An arbitrary argument.

Table 11.2 Methods Defined in the Events Class

Method	Description
translate(int, int)	Translates an event relative to the given component. This involves translating the coordinates so they make sense within the given component.
shiftDown()	Checks to see if the Shift key is pressed; returns true if it is pressed.
controlDown()	Checks to see if the Control key is pressed; returns true if it is pressed.
ToString()	Returns the string representation of the event's values.
metaDown()	Checks to see if the meta key is pressed. Returns true if it is pressed. The meta key is different for each operating system. On a PC, the meta key is the Alt key and on a Mac, the meta key is the Apple key.

correspond to certain events and conditions. We will use these values to compare events that occur in our applications so that we can tell what event has occurred.

Mouse Events

Now that you are familiar with the **Event** class, let's look at some of the methods that are triggered when an event happens. The first ones we'll discuss are the mouse events. These events are probably the most common ones that you will need to check for. The methods for processing these events can be placed in several different places in your program. At the highest level, you can override the events of the GUI elements themselves. For example, you can create your own button class by extending the **Button** class. Then, you can override the default mouse events with your own code. The next option is to place a button or multiple buttons on a panel and override a button's mouse events. With this scenario, you must use **if** or **switch** statements to detect which button is pressed.

The final option is to override the mouse events of the applet or frame you are using for the entire program. This method gets difficult when you have complex UI environments. Let's take a close look at each of the mouse events in detail.

mouseDown()

Clicking on a mouse button creates two distinct events, **mouseDown** and **mouseUp**. The **mouseDown** event occurs when a button is initially pressed and the **mouseUp** event occurs when a button is released. Why are two events required? Often, you will want to perform different tasks when a button is pressed and when it is released. For example, consider a standard screen button. If you press a mouse button while you are over the screen button, the button should look like it has been depressed. The button would remain in this "down" state until the mouse button is released.

The **mouseDown**() method accepts three arguments:

```
public boolean mouseDown(Event, int, int) {}
```

The first argument is an **Event** object that holds all the information about the event that has occurred. The second and third arguments are the x and y coordinates representing where the event took place. The values stored in these arguments are the same as the values stored in the **x** and **y** variables found in the **Events** class.

Here is an example that uses the **mouseDown**() method. It illustrates that the x,y coordinate values set by this method are the same as the values stored in the x,y instance variables contained in the **Events** class:

```
import java.awt.*;

class testEvents extends Frame {
   Panel P1;

   testEvents() {
      super("Test Events");
      P1 = new Panel();
      setLayout(new FlowLayout());
      P1.setBackground(new Color(255,255,255));
      add(P1);
      resize(300,200);
      show();
   }

   public boolean mouseDown(Event evt, int x, int y) {
      System.out.println("X, Y = " + x + ", " + y);
      System.out.println("Event X, Y = " + evt.x + ", " + evt.y);
      return true;
   }

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

MOUSEUP()

The **mouseUp**() event method is implemented in the exact same way as the **mouseDown**() event method. When you are creating routines that respond to simple mouse clicks this is usually the place to put the code. Why here instead of in a **mouseDown** event method? Well, think about how people use an interface. Is it more natural for a mouse click event to occur the instant the button is pressed, or when it is released? If you look at how other programs work, you will notice that most, if not all, don't respond until the mouse button is released.

You should follow this paradigm for two reasons. First, it represents the standard way of processing mouse clicks and you do not want to create an interface that seems inconsistent to the user. Second, it gives the user an opportunity to change his or her mind. After all, how many times have you started to press a button in a program only to change your mind, move the mouse off the button, and *then* let go?

Here is the declaration for the **mouseUp**() method:

```
public boolean mouseUp(Event, int, int) {}
```

Once again, we are given three arguments—an **Event** object, and two integers that give us the x and y coordinates of the event.

mouseMove() and mouseDrag()

The **mouseMove**() event method is used to constantly give feedback when the mouse pointer is over a component. The **mouseDrag**() event method tells us the same thing, but only while one of the mouse buttons is pressed. Whenever the mouse is being moved over your component, one of these methods will constantly be called. Be careful not to put code in these methods that takes too long to execute. If you do, you may see some performance degradation in your program.

Here are the declarations for the **mouseMove**() and **mouseDrag**() event methods:

```
public boolean mouseMove(Event, int, int) {}
public boolean mouseDrag(Event, int, int) {}
```

Again, three arguments are used; an **Event** object, and two integers that represent the x and y coordinates where the event occurs.

Here is a very simple program that responds to mouse movement and dragging by displaying the location of the event in the title bar:

```
import java.awt.*;

class testEvents extends Frame {

   testEvents() {
      super("Test Events");
      resize(300,200);
      show();
   }

   public boolean mouseMove(Event evt, int x, int y) {
      setTitle("mouseMove at: " + x + ", " + y);
      return true;
   }

   public boolean mouseDrag(Event evt, int x, int y) {
      setTitle("mouseDrag at: " + x + ", " + y);
      return true;
   }

   public static void main(String args[]) {
      testEvents TE = new testEvents();
```

```
    }
}
```

As you can see, it is extremely easy to respond to mouse events. You may notice that in this example and in the previous one, you cannot exit out of the program. Basically, what you need to do is check for another event, a system event that tells you that the user is trying to close the window. This is a very easy thing to look for, but we did not want to confuse the code with extra methods. Later in this chapter we'll show you how to check for system events.

mouseEnter() and mouseExit()

These two events come in handy for certain situations. For example, if you want to provide feedback to the user when he or she moves the mouse pointer into or out of your components, you may want to display a message in a status bar. You can get basically the same effect by checking for the **mouseMove**() method, but this method gets called many times while the mouse is over a component and the **mouseEnter**() and **mouseExit**() methods get called only once. The declarations for these event methods are:

```
public boolean mouseEnter(Event, int, int) {}
public boolean mouseExit(Event, int, int) {}
```

These methods are also useful for keeping track of how long a person keeps their mouse over a certain component. For example, assume you were creating a game and you wanted to cause an event to occur if the player keeps the pointer over the "fire" button for too long. You could then respond with a sound or message. Here is an example that checks the time at which the user moves the mouse onto the applet and if the user stays for more than two seconds, the status bar displays an error message. When the user leaves the applet space and returns, the message returns to normal.

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

// testNav Class
public class testTime extends Applet {
   Button B1;
   long downTime;
```

```
   public testTime() {
      setLayout(new FlowLayout());
      B1 = new Button("Click Me!");
      add("Center", B1);
   }

   public boolean mouseEnter(Event evt, int x, int y) {
      downTime = evt.when;
      return true;
   }

   public boolean mouseExit(Event evt, int x, int y) {
      downTime = 0;
      return true;
   }

   public boolean mouseMove(Event evt, int x, int y) {
      if ((evt.when - downTime) > 2000) {
         B1.setLabel("Too Long!");
      } else {
         B1.setLabel("Click Me!");
      }
      return true;
   }

   public void init(){
      testTime TT = new testTime();
   }

} // End testNav
```

Keyboard Events

Handling keyboard events is similar to handling mouse events. The big difference is that when you process a mouse event, you have only one mouse button to work. On the other hand, the user can press one of many possible keys, so processing keyboard events requires an extra step. The two methods you can use, **keyDown**() and **keyUp**(), are very similar to the **mouseDown**() and **mouseUp**() event methods. The only difference is that the keyboard events *do not* generate a location where the event has occurred. Instead, they generate an integer value representing the key that was pressed.

KEYDOWN() AND KEYUP()

Here are the declaration statements for the **keyDown**() and **keyUp**() event methods:

```
public boolean keyDown(Event, int) {}
public boolean keyUp(Event, int) {}
```

The integer argument stores the numeric value of each key on the keyboard—the ASCII equivalent of a key. Java offers us a simple technique for converting this ASCII value to a character representation; we simply cast the integer to a **char** and there we have it. Here is a simple example:

```
public boolean keyDown(Event evt, int key) {
   System.out.println("Value = " + key + ", character = " + (char)key);
}
```

This little code snippet simply waits for a key to be pressed then prints the numeric value of the key and the character representation of that value.

If you look back at the variables defined in the **Event** class (Table 11.1), you'll notice all of the key representations. These values represent the keys that do not belong to the standard ASCII character set, including keys like Home, Page Up, and the function keys. Here's a simple code example that waits for you to press the **F1** key to get help and the Ctrl-x combination to quit the program:

```
import java.awt.*;

class testKeys extends Frame {

   testKeys() {
      super("Test Keys");
      resize(300,200);
      show();
   }

   public boolean keyDown(Event evt, int key) {
      if (evt.controlDown()) {
         if (key == 24) {
            System.exit(0);
            return true;
         }
      }
      if (key == Event.F1) {
         setTitle("No help here!");
         return true;
      }
      return false;
   }
```

```
   public boolean keyUp(Event evt, int key) {
      if (key == Event.F1) {
         setTitle("Test Keys");
         return true;
      }
      return false;
   }

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

Hierarchy of Events

Now that you've seen how to create event handlers for specific events, let's look at how you can create event handlers that capture everything in one central location. Why would you want to do that? Assume that you have an applet that uses a set of buttons that perform related operations. You can create separate event handlers for each control individually, or you can create an event handler that captures all the events at one location and then uses the information in the **Event** object to determine what happened to a certain control.

If you recall from Chapter 9, we created an applet with four buttons. Each button had a caption that represented a URL for a Web site. When a button is pressed, the event handler of the applet, not the buttons themselves, processes the event. This "centralized" approach makes the events easier to process; however, sometimes it can get a little crowded, especially if you have many components that must respond differently to certain events. In these cases it is up to you to decide where to handle the events.

If you do not handle an event at the component level, make sure that a system is in place to hand that event off to the container the particular control resides in. Let's create a little program that has two buttons in a frame. In the first example we will use a standard button and catch its events with the frame. In the second example we will create our own button by subclassing the **Button** class and adding our own event handler to that button. Here is the first example:

```
import java.awt.*;

class testEvents extends Frame {
   myButton B1;
   myButton B2;
```

```
   testEvents() {
      super("Test Keys");
      setLayout(new FlowLayout());
      add(new Button("Hello!");
      add(new Button("Goodbye!");
      resize(300,200);
      show();
   }

   public boolean mouseDown(Event evt, int x, int y) {
      if (evt.target instanceof Button) {
         System.out.println((String)evt.arg);
         return true;
      } else {
         return super.mouseDown(evt, x, y);
      }
   }

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

In the second example we create our own event handler:

```
import java.awt.*;

class testEvents extends Frame {
   myButton B1;
   myButton B2;

   testEvents() {
      super("Test Keys");
      setLayout(new FlowLayout());
      B1 = new myButton("Hello!");
      B2 = new myButton("Goodbye!");
      add(B1);
      add(B2);
      resize(300,200);
      show();
   }

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

```
class myButton extends Button {

   myButton(String s) {
      super(s);
   }

   public boolean mouseDown(Event evt, int x, int y) {
      System.out.println((String)evt.arg);
      return true;
   }
}
```

Processing System Events

Not only can you decide where to place your event handlers, Java also gives you options for capturing events. Instead of creating separate event handlers for each type of event, you can also create a single centralized event handler that accepts all events and then figures out what they are and which components are responsible for generating them.

For processing system-related events, Java provides **handleEvent**() and **action**(). The **action**() method captures all of the standard mouse and keyboard events. The **handleEvent**() method handles all of those and more. It can also catch all the system messages that might be sent to your program.

If you have several different methods in your code that all catch the same event, you need to know what order the methods will be fired in. For example, assume you have a class that has a **mouseDown**() method, an **action**() method, and a **handleEvent**() method. If the user clicks the mouse button, which method will get called? The first method to get called will be the **handleEvent**() method—the mother of all event handlers. If it does not handle the event, the event will be passed on to the **action**() method. Finally, if the **action**() method does nothing with the event, the event is passed on to the **mouseDown**() method.

With this knowledge in hand, you can better direct the flow of operations in your code. You can even handle events in multiple locations. The key here is to do something with the event, then pass it on as if it were never processed. In the case of the event methods, you would usually return a value of true at the end of a method where you handled an event. If you change the return value to false, you will see that the event continues down the chain. Here is a sample applet that illustrates this technique:

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

public class testEvents4 extends Applet {
   Label L1 = new Label();
   Label L2 = new Label();
   Label L3 = new Label();

   public testEvents4() {
      setLayout(new GridLayout(2, 3));
      add(new Label("handleEvent()"));
      add(new Label("action()"));
      add(new Label("mouseMove()"));
      add(L1);
      add(L2);
      add(L3);
   }

   public boolean handleEvent(Event evt) {
      if (evt.id == Event.MOUSE_MOVE) {
         L1.setText("" + evt.when);
         this.action(evt, evt.target);
         return super.handleEvent(evt);
      } else {
         return false;
      }
   }

   public boolean action(Event evt, Object arg) {
      L2.setText("" + evt.when);
      return true;
   }
   public boolean mouseMove(Event evt, int x, int y) {
      L3.setText("" + evt.when);
      return true;   }

   public void init(){
      testEvents TN = new testEvents();
   }
} // End Application
```

Try switching some of the return statements and see what happens. Figure 11.1 shows the applet in action. The applet consists of six **Label** controls. They are arranged in a grid. The top three labels list the three methods we are handling. The three lower labels show the time at which the last event of the respective type took place.

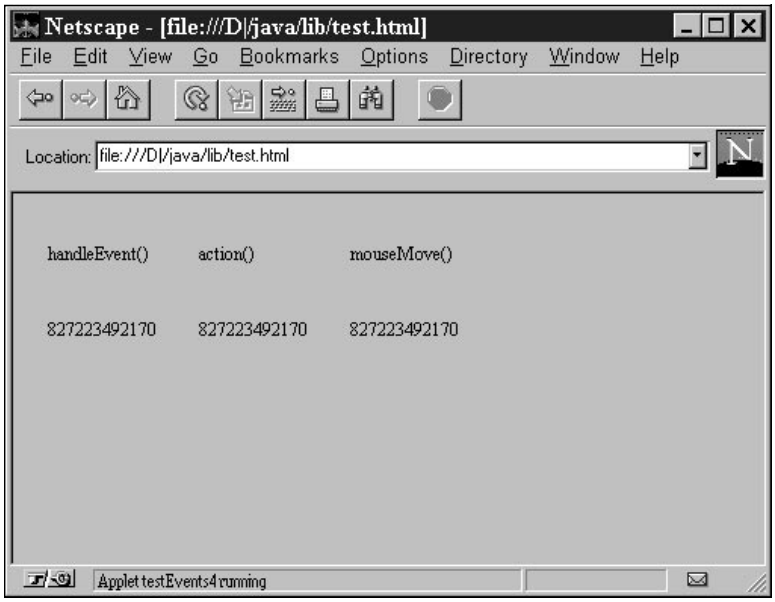


Figure 11.1
The sample events applet.

ACTION() METHOD

The **action**() method is useful for responding to a multitude of user actions. It will capture any events that are caused directly by the user. For example, it will catch mouse clicks, but it will not catch system calls. Here is the declaration for the **action**() method:

```
public boolean action(Event, Object) {}
```

HANDLEEVENT() METHOD

The **handleEvent**() method is probably the most versatile of the event handling methods. It can catch all the standard user interface events as well as system events. The ability to catch system events is crucial to larger, more complex programs. It is also essential to applications. Without the ability to catch system events, applications would never know when the user has pressed any of the title bar buttons or selected any of the title bar options.

Here is the declaration for the **handleEvent**() method:

```
public boolean handleEvent(Event) {}
```

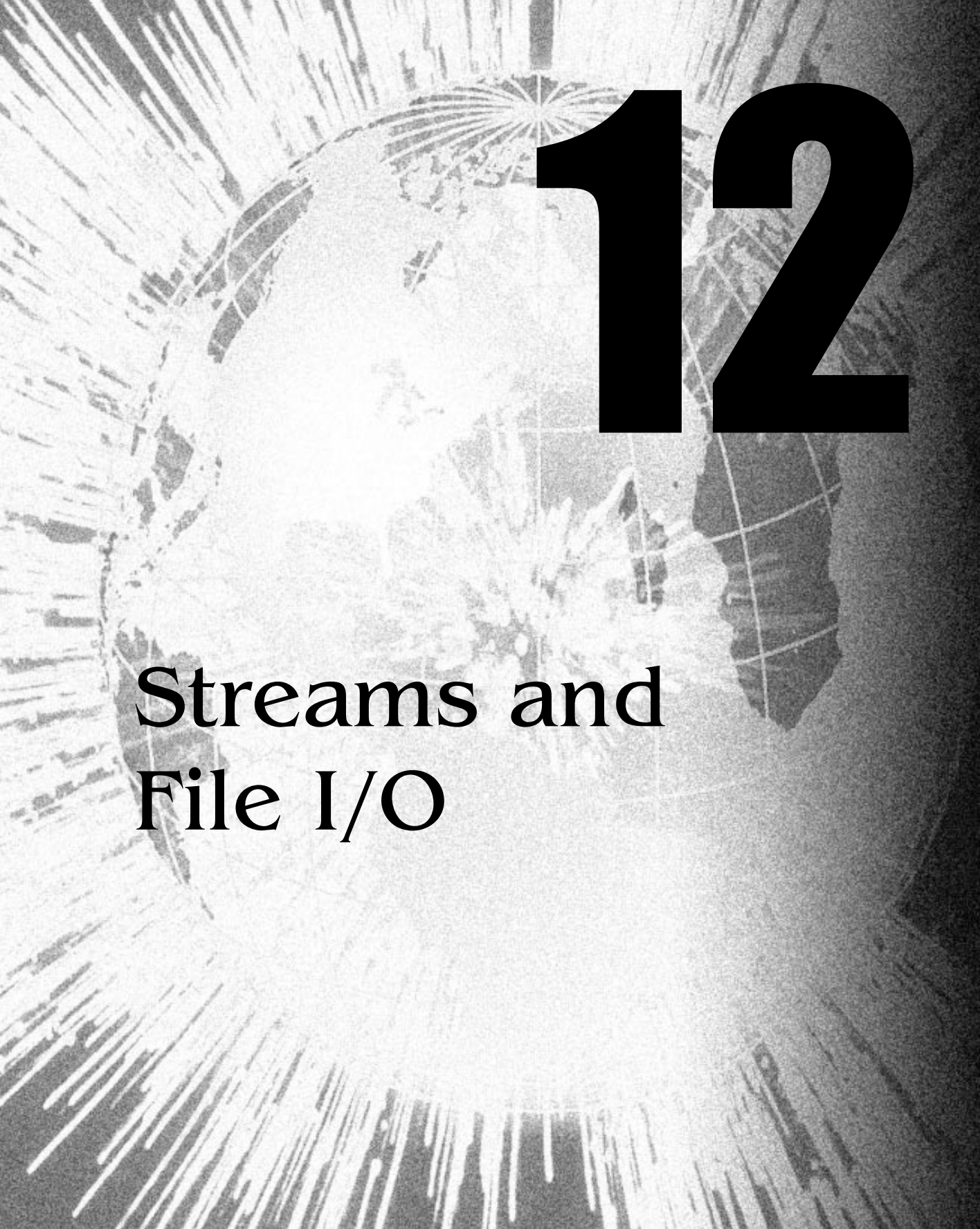
You probably have seen this method in use in several of our example programs. When the user of an application clicks on the close icon for your application, it sends a message indicating that the close icon has been pressed. This message comes in the form of **WINDOW_DESTROY**. We need to catch this event and then tell the application to quit. Here is a code snippet that will do just that:

```
public boolean handleEvent(Event evt) {
   if (evt.id == Event.WINDOW_DESTROY) {
      System.exit(0);
      return true;
   } else {
      super. HandleEvent(evt);
      return false;
   }
}
```

This simply checks the event **id** variable to see if it equals the static variable **WINDOW_DESTROY** defined in the **Event** class. If we receive this event message, we tell the application to close by calling the **exit**() method of the **System** class.

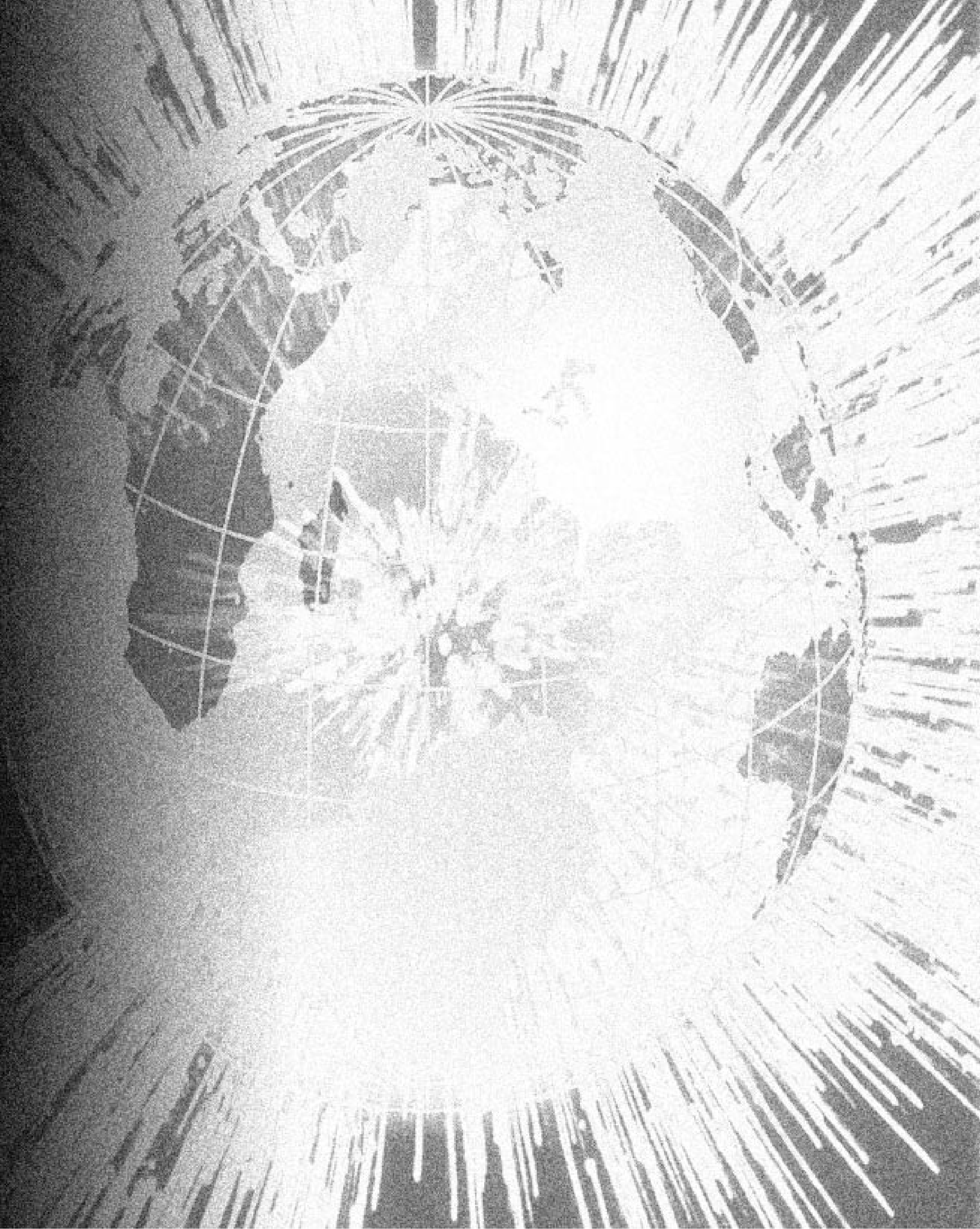
Problems with Processing Events

Java does not handle all events very well yet. In particular, the current Java interpreters often make mistakes or do not respond correctly to all events. The most common errors involve the mouse event methods. Sometimes the parent objects do not receive any stimulation when they should. For example, if you place a button in the middle of an applet, you should be able to catch all the mouse events that happen to the button with an event handler belonging to the applet. This *should* work, but in practice it is a little flaky.



12

Streams and File I/O



12 Streams and File I/O

It's now time to learn the ins and outs of how streams are used in Java to perform a variety of I/O operations.

What good is a program that has no means of communicating with the outside world? If you think about it, most programs you write follow a simple pattern of getting data from a user, processing the data, and presenting the user with the results in one format or another. Java arranges the world of input/output (I/O) into a system of byte *streams*. A byte stream is essentially an unformatted sequence of data which can come from or be sent to a number of different sources including the keyboard, screen, file, and so on. To help you process input and output streams in your programs, Java provides special streams in the **System** class as well as other custom classes including **InputStream** and **OutputStream**.

In this chapter we'll show you how to use the **System** class and the java.io package to perform different types of stream I/O from reading strings typed in at the keyboard to writing data to files. After we introduce the three streams supported by the **System** class we'll examine the other key stream processing classes including **InputStream**, **OutputStream**, **BufferedInputStream**, **BufferedOutputStream**, **ByteArrayInputStream**, **ByteArrayOutputStream**, **DataInputStream**, **DataOutputStream**, **FileInputStream**, **FileOutputStream**, and others.

Introducing the System Class

The **System** class is responsible for providing access to the three main streams: **System.in**, **System.out**, and **System.error**. All input streams are derived from **System.in**, which is responsible for reading data. All of the output streams are derived from **System.out**, which is responsible for sending out data in one form

or another. The last stream is **System.error**, which is an output stream derived from **System.out**. As its name implies, **System.out** handles the errors that occur while I/O operations are performed.

To use the streams implemented by the **System** class, you must import the javio.io package:

```
import java.io.*;
```

Here's a simple program that uses **System.in** and **System.out** to read and write a string of text:

```
import java.io.*;

public class ProcessALine {
   public static void main(String arg[]) {
      // Bring in the stream from the keyboard and pipe it to DataInput
      DataInputStream aDataInput = new DataInputStream(System.in);
      String aString;

      try {
         // Continue to read lines from the keyboard until ^Z is pressed
         while ((aString = aDataInput.readLine()) != null) {
            // Print the line out to the screen
            System.out.println(aString);
         }
       } catch (IOException e) { // Check for I/O errors
            System.out.println("An IOException has occurred");
       }
   }
}
```

The **System.in** stream is used to create an input stream object called **aDataInput**. This object is created as an instance of the **DataInputStream** class. The advantage of this class is that it contains methods like **readLine**() for processing input streams. In fact the work of reading a string is accomplishing using a single line of code:

```
while ((aString = aDataInput.readLine()) != null)  ...
```

The **readLine**() method will continue to read characters from the input stream until a Ctrl-Z character is encountered, which terminates the input stream. (Thus, when you run this program, make sure that you enter a Ctrl-Z at some point to tell the program to stop reading characters from the input stream.)

Checking for Errors

One important feature you'll find in our sample program is simple error handling code implemented with the **try** ... **catch** clause. When performing I/O stream operations, you must check for possible I/O errors. Notice that in our program, a **try** clause is used to check *both* the stages of reading and writing to a stream. If an error occurs, an **IOException** error is thrown and the **catch** clause will be executed.

Different Flavors of Streams

With the multitude of possible forms your data can come in and be sent out with, it's no wonder that Java offers a variety of stream handlers that can be implemented. Each stream type is capable of combining other stream types to handle unique situations. For example, if you want to buffer data that is being read from a file so that it may be read all at once, you could use a statement like this:

```
InputStream aStream =
   new BufferedInputStream(new FileInputStream("C:\foobar.txt"));
```

This line would bring in the data from a file, buffering it in a stream called **aStream**. Then, the data could be accessed all at once by using **aStream**. This technique could be very valuable in applications that need to read data all at once instead of having to read chunks and perform multiple read operations.

Understanding the Basics of Inputting and Outputting

All data that you manipulate in the form of strings and numbers, be it an integer or a double, must be transformed into a stream (bytes) in order for the information to be interpreted by the computer and the devices that use the data. This is considered *outputting* the data verses *inputting* the data. Likewise, if a user needs to understand the data, unless they read machine language, we need to convert it to a useful format by manipulating it with one of the input streams. Figure 12.1 shows the process of how data is converted and processed with both input and output streams.

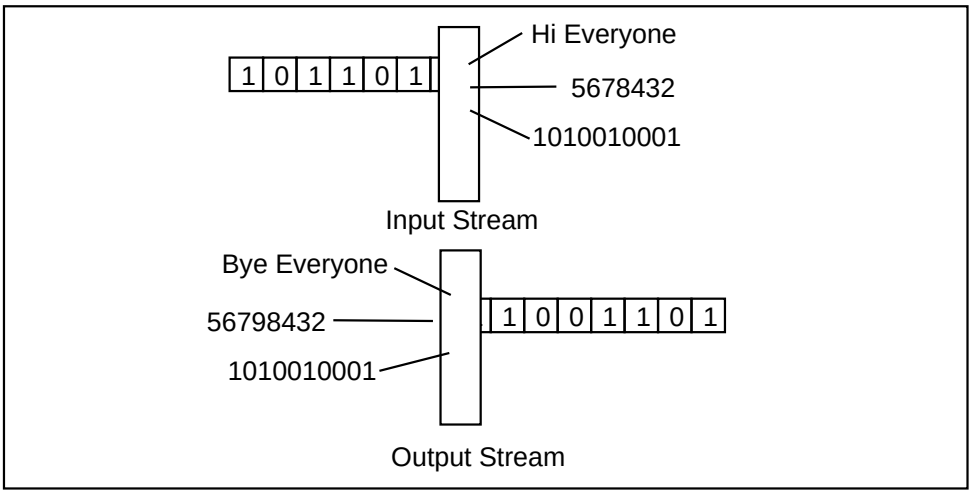


Figure 12.1
A graphical example of converting data for input and output streams.

InputStream and OutputStream Classes

In addition to the **System** class, Java provides other classes for handling stream input and output including **InputStream** and **OutputStream**, which are abstract classes responsible for the basic declarations of all the input and output streams that we'll be exploring next. All streams created from these classes are capable of throwing an **IOException** that must be dealt with by "re-throwing" it or handling it. To declare and create an object using **InputStream** or **OutputStream** you would use statements like the following:

```
InputStream aStream = getStreamFromASource();
OutputStream aStream = getStreamToASource();
```

In both declarations, the methods *getStreamFromSource()* and *getStreamToASource()* are any methods that returns a stream. The stream returned by the method will be assigned to the object *aStream*. All information passed to and from the streams must be in the form of bytes. Table 12.1 presents the key methods defined in **InputStream** and Table 12.2 presents the methods defined in **OutputStream**.

Table 12.1 Key Methods Defined in the InputStream Class

Method	Description
available()	Returns the number of bytes available in the stream without invoking a block.
close()	Closes the stream.
mark(int)	Marks the current position in the stream.
markSupported()	Returns a true/false value to indicate if the stream supports marking capabilities.
read()	Reads bytes from the stream.
read(byte[])	Reads bytes from the stream and places them in an array.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
reset()	Sets the stream to the last mark.
skip(long)	Skips a designated number of bytes in the stream.

Table 12.2 Key Methods Defined in the OutputStream Class

Method	Description
close()	Closes the stream.
flush()	Clears the stream, forcing any buffered bytes to be written out.
write(int)	Writes *n* number of bytes to the stream.
write(byte[])	Writes an array of bytes to the stream.
write(byte[], int, int)	Writes an array of bytes of a specified size, starting at a specified index position.

Here's an example of an application that uses the **OutputStream** class to derive another class for performing a simple output operation:

```
import java.io.*;
import java.net.*;
import java.awt.*;

public class mrsServer extends Frame {
   TextArea serverScreen;

    mrsServer() {
        super("Server Application");
        serverScreen = new TextArea("mrsServer's Screen:\n", 10, 40);
        add("Center", serverScreen);
        pack();
        show();
```

```
        ServerSocket mrsServer = null;
        Socket socketReturn = null;
        // Assigns the variable rawDataOut to the class OutputStream
        OutputStream rawDataOut = null;

        try {
            mrsServer = new ServerSocket( 10, 2 );
            socketReturn = mrsServer.accept();
            serverScreen.appendText( "Connected to the mrsServer" );

            // Creates an instanceof the class OutputStream named
            // rawDataOut
            // rawDataOut receives the stream from the Socket socketReturn
            rawDataOut = socketReturn.getOutputStream();
            DataOutputStream DataOut = new DataOutputStream(rawDataOut);

            DataOut.write( 5 );
        } catch( UnknownHostException e ) {
        } catch( IOException e ) {
        }
    }
```

BufferedInputStream and BufferedOutputStream Classes

These classes are used to implement streams responsible for collecting data from a source that brings in data at a slower rate than the recipient. Then, the chunks of data collected may be delivered in larger, more manageable blocks than in the manner they were received. This proves beneficial for file systems and networks, where the connection depends on a device for transmission.

Here are the declarations for these classes:

```
InputStream aStream = new BufferedInputStream(getStreamFromASource());
OutputStream aStream = new BufferedInputStream(getStreamToASource());
```

In both declarations, a stream is returned to the parent class, **InputStream** or **OutputStream** depending if it is incoming or outgoing. The methods *getStreamFromSource()* and *getStreamToASource()* can be any methods that return a stream. The stream that the method returns will be buffered as the information comes in and is appended to the object **aStream**. All information passed to and from the streams must be in the form of bytes. Table 12.3 presents the

Table 12.3 Key Methods Defined in the BufferedInputStream Class

Method	Description
available()	Determines the number of bytes available in the stream without invoking a block.
mark(int)	Marks the current position in the stream.
markSupported()	Returns a true/false value to indicate if the stream supports marking capabilities.
read()	Reads bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
reset()	Sets the stream to the last mark.
skip(long)	Skips a designated number of bytes in the stream.

Table 12.4 Key Methods Defined in the BufferedOutputStream Class

Method	Description
flush()	Clears the stream, forcing any buffered bytes to be written out.
write(int)	Writes *n* number of bytes to the stream.
write(byte[], int, int)	Writes an array of bytes of a specified size, starting at a specified index position.

key methods defined in **BufferedInputStream** and Table 12.4 presents the methods defined in **BufferedOutputStream**.

Here's an example of an application that uses the **BufferedInputStream** class to derive another class for performing a simple input operation:

```
import java.io.*;

// Reads from a file
public class  ReadAFile extends Object {
   ReadAFile(String s) {
      String line;
      FileInputStream fileName  = null;
      // Assigns the variable bufferedInput to the class
      // BufferedInputStream
      BufferedInputStream bufferedInput = null;
      DataInputStream dataIn = null;

      try {
         fileName = new FileInputStream(s);
         // Creates an instance of the class BufferedInputStream named
         // bufferedInput
```

```
            // bufferedInput receives the stream from the FileInputStream
            // fileName as it is read
            bufferedInput = new BufferedInputStream(fileName);
            dataIn = new DataInputStream(bufferedInput);
         }
         catch(FileNotFoundException e) {
            System.out.println("File Not Found");
            return;
         }
         catch(Throwable e) {
            System.out.println("Error in opening file");
            return;
         }

         try {
            while ((line = dataIn.readLine()) != null) {
               System.out.println(line + "\n");
            }
            fileName .close();
         }
         catch(IOException e) {
            System.out.println("Error in reading file");
         }
      }

      // Where execution begins in a stand-alone executable
      public static void main(String args[]) {
          new ReadAFile(args[0]);
      }
}
```

ByteArrayInputStream and ByteArrayOutputStream Classes

These streams create a new stream from an array of bytes to be processed. They are used to perform the reverse operations of what most of the streams do. To declare and create an object from these classes you would use statements like the following:

```
InputStream aStream = new ByteArrayInputStream(getStreamFromASource());
OutputStream aStream = new ByteArrayInputStream(getStreamToASource());
```

The methods used above, *getByteArrayFromSource()*and *getByteArrayToASource()* can be any methods that returns a byte array. The array is passed through the **ByteArrayInputStream** class and a stream is created from it. The stream that was converted from the class returns a value and is assigned to the object**aStream**. All information passed to and from the streams must be in the form of bytes. Table 12.5 presents the key methods defined in **ByteArrayInputStream** and Table 12.6 presents the methods defined in **ByteArrayOutputStream**.

Here is a hypothetical example that uses a **ByteArrayInputStream** class. If you wanted to compile this program, you would need to supply a method to fill the array of bytes, **anArrayOBytes**:

Table 12.5 Key Methods Defined in the ByteArrayInputStream Class

Method	Description
available()	Returns the number of bytes available in the stream without invoking a block.
read()	Reads bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
reset()	Sets the stream to the last mark.
skip(long)	Skips a designated number of bytes in the stream.

Table 12.6 Key Methods Defined in the ByteArrayOutputStream Class

Method	Description
reset()	Resets the current buffer so that it may be used again.
size()	Returns the current size of the buffer.
toByteArray()	Returns a copy of the input data.
toString()	Converts input bytes to a string.
toString(int)	Converts input bytes to a string, sets the selected byte's first 8 bits of a 16 bit Unicode to hibyte.
write(int)	Writes *n* number of bytes to the buffer.
write(byte[], int, int)	Writes an array of bytes of a specified size, starting at a specified index position.
writeTo(OutputStream)	Writes the buffered information to another stream.

```
import java.io.*;

// Reads from a file
public class  Byte2String extends Object {
   Byte2String(String s) {
      byte[] anArrayOBytes;
      …
      //fills the anArrayOBytes with data
      …
      try {
         // Creates an instanceof the class InputStream named byteDataIn
         // byteDataIn receives the stream from the ByteArrayInputStream
         // anArrayOBytes
         InputStream byteDataIn = new ByteArrayInputStream(anArrayOBytes);
      }
      catch(IOException e) {
      }
      …
      // perform some process with the stream
   }

   // Where execution begins in a stand-alone executable
   public static void main(String args[]) {
       new Byte2String(args[0]);
   }
}
```

DataInputStream and DataOutputStream Classes

All methods defined in these classes are actually declared in an interface named **DataInput**. To declare and create a **DataInputStream** or **DataOutputStream** object you would use statements like the following:

```
DataInputStream aStream = new DataInputStream(getStreamFromASource());
DataOutputStream aStream = new DataOutputStream(getStreamToASource());
```

Notice in both declarations the need to declare the class type, **DataInputStream** or **DataOutputStream**, instead of type **InputStream** or **OutputStream**. The methods *getStreamFromSource()* and *getStreamToASource()* can be any methods that return a stream. Once the stream is passed to the **DataInputStream** or **DataOutputStream** object, the methods declared in the interface can be applied to the stream. Table 12.7 presents the key methods defined in **DataInputStream** and Table 12.8 presents the methods defined in **DataOutputStream**.

Table 12.7 Key Methods Defined in the DataInputStream Class

Method	Description
read(byte[])	Reads an array of bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
readBoolean()	Reads a boolean from the stream.
readByte()	Reads a 8-bit byte from the stream.
readChar()	Reads a 16-bit character from the stream.
readDouble()	Reads a 64-bit double from the stream.
readFloat()	Reads a 32-bit float from the stream.
readFully(byte[])	Reads bytes from the stream, blocking until all bytes are read.
readFully(byte[], int, int)	Reads bytes from the stream, blocking until all bytes are read. The starting point to begin reading and the maximum number of bytes to read are passed as parameters.
readInt()	Reads a 32-bit integer from the stream.
readLine()	Reads a line from the stream until an \n,\r,\n\r \, or EOF is reached.
readLong()	Reads a 64-bit long from the stream.
readShort()	Reads a 16-bit short from the stream.
readUTF()	Reads a UTF formatted string from the stream.
readUTF(DataInput)	Reads a UTF formatted string from a specific stream.
readUnsignedByte()	Reads an unsigned 8-bit byte from the stream.
readUnsignedShort()	Reads an unsigned 8-bit short from the stream.
skipBytes(int)	Skips a designated number of bytes in the stream, blocking until finished.

Table 12.8 Key Methods Defined in the DataOutputStream Class

Method	Description
flush()	Clears the stream, forcing any buffered bytes to be written out.
size()	Returns the number of bytes in the stream.
write(int)	Writes *n* number of bytes to the stream.
write(byte[], int, int)	Writes an array of bytes of a specified size, starting at a specified index position.
writeBoolean(boolean)	Writes a boolean to the stream.
writeByte(int)	Writes an 8-bit byte to the stream.
writeBytes(String)	Writes a string of bytes to the stream.

continued

Table 12.8 Key Methods Defined in the DataOutputStream Class (Continued)

Method	Description
writeChar(int)	Writes a 16-bit character to the stream.
writeChars(String)	Writes a string of chars to the stream.
writeDouble(double)	Writes a 64-bit double to the stream.
writeFloat(float)	Writes a 32-bit float to the stream.
writeInt(int)	Writes a 32-bit integer to the stream.
writeLong(long)	Writes a 64-bit long to the stream.
writeShort(int)	Writes a 16-bit short to the stream.
writeUTF(String)	Writes a UTF formatted string to the stream.

Let's revisit our example of the client application once more to demonstrate the use of a **DataInputStream**:

```
import java.io.*;
import java.net.*;
import java.awt.*;

public class mrClient extends Frame {
    mrClient() {
        super("Client Application");
        TextArea clientScreen = new TextArea("mrClient's Screen:\n", 10, 40);
        add("Center", clientScreen);
        pack();
        show();
        Socket mrClient = null;
        InputStream rawDataIn = null;

        try {
            mrClient = new Socket( InetAddress.getLocalHost(), 10 );

            rawDataIn = mrClient.getInputStream();
            // reads in the stream for the InputStream rawDataIn and pipes
            // it to DataIn
            DataInputStream DataIn = new DataInputStream(rawDataIn);
            // the array of bytes is then read from the stream
            clientScreen.appendText( "mrClient receives -  " +
            DataIn.read() );
        } catch( UnknownHostException e ) {
        } catch( IOException e ) {
        }
    }
```

Here the stream used with the **Socket** object, **mrClient**, is piped from the **InputStream** to the **DataInputStream**. The bytes are then read from the stream, **DataIn**, and appended to the text box with this line:

```
clientScreen.appendText("mrClient receives -  " + DataIn.read());
```

By simply changing the method, we can read any type of data that resides in the stream. For example, if we wanted to read a stream of chars, we would use a statement like this:

```
clientScreen.appendText("mrClient receives -  " + DataIn.readChar());
```

FileInputStream and FileOutputStream Classes

The most common use for these streams is to apply them to a file to be read from and written to. Here are the declarations for these classes.

```
InputStream aStream = getStreamFromASource();
OutputStream aStream = getStreamToASource();
```

In both declarations, the methods *getStreamFromSource()* and *getStreamToASource()* can be any methods that return a stream. The stream that the method returns is assigned to the object **aStream**. All information passed to and from the streams must be in the form of bytes. Table 12.9 presents the key methods defined in **FileInputStream** and Table 12.10 presents the methods defined in **FileOutputStream**.

Table 12.9 Key Methods Defined in the FileInputStream Class

Method	Description
available()	Returns the number of bytes available in the stream without invoking a block.
close()	Closes the stream.
finalize()	Closes the stream when the garbage collector is invoked.
getFD()	Returns the file descriptor of the file associated with the stream.
read()	Reads bytes from the stream.
read(byte[])	Reads into an array of bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
skip(long)	Skips a designated number of bytes in the stream.

Table 12.10 Key Methods Defined in the FileOutputStream Class

Method	Description
close()	Closes the stream.
finalize()	Closes the stream when the garbage collector is invoked.
getFD()	Returns the file descriptor of the file associated with the stream.
write(int)	Writes *n* number of bytes to the stream.
write(byte[])	Writes an array of bytes to the stream.
write(byte[], int, int)	Writes an array of bytes of a specified size, starting at a specified index position.

Here is a practical example of how you can use the **FileOutputStream** and **FileInputStream** classes:

```
import java.io.*;

public class WriteAFile {
   WriteAFile(String s) {
      write(s);
   }

   // Writes to a file
   public void write(String s) {
      // Assigns the variable writeOut to the class FileOutputStream
      FileOutputStream writeOut = null;
      DataOutputStream dataWrite = null;

      try {
         // Creates an instanceof the class FileOutputStream named writeOut
         // writeOut receives the stream from the File designated in the
         // variables
         writeOut = new FileOutputStream(s);
         dataWrite = new DataOutputStream(writeOut);
         dataWrite.writeChars("This is a Test");
         dataWrite.close();
      }
      catch(IOException e)  {
         System.out.println("Error in writing to file");
       }
      catch(Throwable e)  {
         System.out.println("Error in writing to file");
       }
      finally {
         System.out.println("\n\n.....creating a backup file.");
```

```
        try {
           // Recreates an instanceof the class FileOutputStream named
           // writeOut
           // writeOut receives the stream from the File named
           // "MyBackup.sav"
           writeOut = new FileOutputStream("MyBackup.sav");
           dataWrite = new DataOutputStream(writeOut);
           dataWrite.writeChars("This is a Test");
           dataWrite.close();
        }
        catch (IOException e) {
                 System.out.println("Error in writing backup file");
         }
      }
   }
   // Where execution begins in a stand-alone executable
   public static void main(String args[]) {
       new WriteAFile(args[0]);
   }
}
```

The variable **writeOut**, which is of type **DataOutputStream**, is actually used twice in this example: once to write the file specified by the user and again to write a file MyBackup.sav.

FilterInputStream and FilterOutputStream Classes

These are classes that act as channels for streams to be passed through. As a stream is passed through the shell, a hierarchy of stream containers are created to perform some processing of bytes as the methods are passed along with it. This structure allows for a chaining effect of shells to break up a complicated task into small steps. Here are the declarations for these classes:

```
FilterInputStream anotherStream = new FilterInputStream(aStream);
FilterOutputStream anotherStream = new FilterOutputStream(aStream);
```

In both declarations, the methods *FilterInputStream()* and *FilterOutputStream()* require that a stream be passed to each method. In return, a stream is assigned to the object named **anotherStream**. Table 12.11 presents the key methods defined in **FilterInputStream** and Table 12.12 presents the methods defined in **FilterOutputStream**.

Table 12.11 Key Methods Defined in the FilterInputStream Class

Method	Description
available()	Returns the number of bytes available in the stream without invoking a block.
close()	Closes the stream.
finalize()	Closes the stream when the garbage collector is invoked.
getFD()	Returns the file descriptor of the file associated with the stream.
read()	Reads bytes from the stream.
read(byte[])	Reads an array of bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
skip(long)	Skips a designated number of bytes in the stream.

Table 12.12 Key Methods Defined in the FilterOutputStream Class

Method	Description
close()	Closes the stream.
flush()	Clears the stream, forcing any bytes to be written out.
write(int)	Writes *n* number of bytes to the stream.
write(byte[])	Writes an array of bytes to the stream.
write(byte[], int, int)	Writes an array of bytes of a specified size, starting at a specified index position.

Here is an example of how a **FilterOutputStream** class can be manipulated at different stages without actually changing the original stream from the **OutputStream**:

```
import java.io.*;
import java.net.*;
import java.awt.*;

public class mrsServer extends Frame {
   TextArea serverScreen;

    mrsServer() {
       ... // perform functions previous to opening the socket
        try {
            mrsServer = new ServerSocket( 10, 2 );
            socketReturn = mrsServer.accept();
```

```
            OutputStream stageOneDataOut = socketReturn.getOutputStream();
            FilterOutputStream stageTwoDataOut = new
            FilterOutputStream(stageOneDataOut);
            // perform some operations on stageTwoDataOut stream
            ...
            FilterOutputStream stageThreeDataOut = new
            FilterOutputStream(stageTwoDataOut);
            // perform some operations on stageThreeDataOut stream
            ...
            FilterOutputStream stageFourDataOut = new
              FilterOutputStream(stageThreeDataOut);
            // write the data from stageFourDataOut
            ...

        } catch( UnknownHostException e ) {
        } catch( IOException e ) {
        }
        ...
    }
```

LineNumberInputStream Class

This class allows for line numbering of each line processed through the stream. It is useful for determining which lines errors have occurred on. To declare and create an object from this class, you use a statement like the following:

```
LineNumberInputStream aStream =
  LineNumberInputStream(getStreamFromASource());
```

In the declaration above, the method *getStreamAFromSource()* retrieves a source stream that is assigned line numbers. The stream that the method **LineNumberInputStream**() returns is assigned to the object **aStream**. Table 12.13 presents the key methods defined in **LineNumberInputStream**.

Here is a real world example of how the **LineNumberInputStream** class can be used:

```
import java.io.*;

// Reads from a file
public class  ReadAFile extends Object {
   ReadAFile(String s) {
      String line;
      FileInputStream fileName  = null;
      // Assigns the variable bufferedInput to the class
```

```
BufferedInputStream
      BufferedInputStream bufferedInput = null;
      DataInputStream dataIn = null;

      try {
         fileName = new FileInputStream(s);
         // Creates an instanceof the class LineNumberInputStream named
         // parsedData
         // parsedData receives the stream from the FileInputStream
         // fileName as it is read
         LineNumberInputStream parsedData = new
         LineNumberInputStream(fileName);
         dataIn = new DataInputStream(parsedData);
      }
      catch(FileNotFoundException e) {
         System.out.println("File Not Found");
         return;
      }
      catch(Throwable e) {
         System.out.println("Error in opening file");
         return;
      }

      try {
         while ((line = dataIn.readLine()) != null) {
            // adds the current line number to the beginning of every line
            System.out.println(parsedData.getLineNumber() + ": " + line +
              "\n");
         }
     fileName .close();
      }
      catch(IOException e) {
         System.out.println("Error in reading file");
      }
   }

   // Where execution begins in a stand-alone executable
   public static void main(String args[]) {
       new ReadAFile(args[0]);
   }
}
```

As the stream is passed to the **parsedData** stream, a line number is assigned to each line in the stream. The line number is then added to the line before printing the line to the browser.

Table 12.13 Key Methods Defined in the LineNumberInputStream Class

Method	Description
available()	Returns the number of bytes available in the stream without invoking a block.
getLineNumber()	Returns the current line number of the stream.
mark(int)	Marks the current position in the stream.
read()	Reads bytes from the stream
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
reset()	Sets the stream to the last mark.
setLineNumber(int)	Sets the current line number.
skip(long)	Skips a designated number of bytes in the stream.

PipedInputStream and PipedOutputStream Classes

These classes allow for pipe-like connection between two threads to allow for safe communication between a shared queue. For this technique to be effective, both threads must implement the class. To declare and create objects from these class, you use statements like the following:

```
PipedInputStream aThreadStreamIn =
  PipedInputStream(getStreamFromASource());
PipedOutputStream aThreadStreamOut = PipedOutputStream(aThreadStreamIn);
```

In both declarations, they must be implemented in the threads to ensure a data stream is not being written to by the other thread. The stream that the method returns is assigned to an **aThreadStreamIn** or **aThreadStreamOut** object. Table 12.14 presents the key methods defined in **PipedInputStream** and Table 12.15 presents the methods defined in **PipedOutputStream**.

Table 12.14 Key Methods Defined in the PipedInputStream Class

Method	Description
close()	Closes the stream.
connect(PipedOutputStream)	Connects the stream to a **PipedOutputStream** of the sender.
read()	Reads bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.

Table 12.15 Key Methods Defined in the PipedOutputStream Class

Method	Description
close()	Closes the stream.
connect(PipedInputStream)	Connects the stream to a PipedInputStream of the intended recipient.
write(int)	Writes *n* number of bytes to the stream.
write(byte[], int, int)	Writes an array of bytes of a specified size, starting at a specified index position.

PrintStream Class

This class is most commonly used to create an instance of the **System** class, where the methods **print**() and **println**() are referenced as class variables for use in the **System.out** and **System.err** calls. The most common output device for this class is the screen. The declaration for the **PrintStream** class is:

```
PrintStream aStream = new PrintStream( getStreamFromASource());
```

Along with this class accepting the **write**(), **flush**(), and **close**() methods, it supports a slew of print methods that will handle just about every I/O operation you will need to perform. Most often, an object created from this class will be used like this:

```
System.out.print(aStream);
System.out.println(aStream);
```

The only difference between the two calls is that the second call appends a return character to the end of the stream. Table 12.16 presents the key methods defined in **PrintStream**. Here is a simple example of how the **PrintStream** class can be used to return a line that was typed in by the user:

```
import java.io.*;

public class ProcessALine {
   public static void main(String arg[]) {
      DataInputStream aDataInput = new DataInputStream(System.in);
      String aString;

      try {
      // A Control Z exits
        while ((aString = aDataInput.readLine()) != null) {
```

```
            System.out.println(aString);
          }
        } catch (IOException e) {
            System.out.println("An IOException has occurred");
        }
    }
}
```

Table 12.16 Key Methods Defined in the PrintStream Class

Method	Description
checkError()	Flushes the stream and returns a boolean in the event of an error.
close()	Closes the stream.
flush()	Clears the stream, forcing any bytes to be written out.
print(Object)	Prints an Object.
print(String)	Prints a String.
print(char[])	Prints an array of chars.
print(char)	Prints a char.
print(int)	Prints an Integer.
print(long)	Prints a long.
print(float)	Prints a float.
print(double)	Prints a double.
print(boolean)	Prints a boolean.
println()	Prints a newline chatacter.
println(Object)	Prints an Object with a newline appended to the end.
println(String)	Prints a String with a newline appended to the end.
println(char[])	Prints an array of chars with a newline appended to the end.
println(char)	Prints a char with a newline appended to the end.
println(int)	Prints an integer with a newline appended to the end.
println(long)	Prints a long with a newline appended to the end.
println(float)	Prints a float with a newline appended to the end.
println(double)	Prints a double with a newline appended to the end.
println(boolean)	Prints a boolean with a newline appended to the end.
write(int)	Writes *n* number of bytes to the stream.
write(byte[], int, int)	Writes an array of bytes, starting point to begin writing, *n* number of bytes to write.

PushbackInputStream Class

This class causes the stream to reaccept a byte that was passed to it by the **InputStream**. By forcing the byte back to the delivering **InputStream**, you can reread the byte as if it had never been read. To declare a stream as **PushbackInputStream** and instaniate it, you could type the following:

```
PushbackInputStream aStream =
  new PushbackInputStream (getStreamFromASource());
```

The *getStreamFromASource()* method can be any method that returns a stream. The stream that the method is assigned to the object **aStream**. All information passed to and from the streams must be in the form of bytes. Table 12.17 presents the key methods that are defined in **PushbackInputStream**.

SequenceInputStream Class

This class allows for two streams to be seamlessly joined. This is especially useful when creating an exception that would pick up where it left off last in a transfer. To declare a stream as type **SequenceInputStream** and instaniate it, you would type the following:

```
InputStream aStream = new SequenceInputStream(firstStream, secondStream);
```

In the declaration, **firstStream** is appended to the **secondStream** to create a single seamless stream, **aStream**. Table 12.18 presents the key methods defined in **SequenceInputStream**.

Table 12.17 Key Methods Defined in the PushbackInputStream Class

Method	Description
available()	Returns the number of bytes available in the stream without invoking a block.
markSupported()	Returns a true/false value to indicate if the stream supports marking capabilities.
read()	Reads bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
unread(int)	Returns a char to the stream as if it had not been read in the first place.

Table 12.18 Key Methods Defined in the SequenceInputStream Class

Method	Description
close()	Closes the stream.
read()	Reads bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.

StringBufferInputStream Class

This class is very similiar to the **ByteArrayInputStream** class. The difference is that it combines an array of char types into a stream. Note, an array of chars is actually a string. To declare and create an object from this class, you use a statement like the following:

```
InputStream aStream = new StringBufferInputStream(String);
```

The classes declared here passes a string through the **StringBufferInputStream** class and a stream is created from it. The stream that was converted from the class returns a value and is assigned to the object **aStream**. Table 12.19 presents the key methods defined in **StringBufferInputStream**.

Here is another hypothetical example, but this time a **StringBufferInputStream** class is used. (You may have recognized this example earlier when we introduced the **ByteArrayInputStream** class. We decided to reuse the sample program because of the similarity between the two classes.)

Table 12.19 Key Methods Defined in the StringBufferInputStream Class

Method	Description
available()	Returns the number of bytes available in the stream without invoking a block.
read()	Reads bytes from the stream.
read(byte[], int, int)	Reads a specified number of bytes from the stream and places them in an array starting at a specified index position.
reset()	Sets the stream to the last mark.
skip(long)	Skips a designated number of bytes in the stream.

```
import java.io.*;

// Reads from a file
public class  Char2Stream extends Object {

 Char2Stream(String s) {
      String aCommonString = "The quick brown fox jumped over the lazy dog";
      try {
         // Creates an instanceof the class InputStream named charDataIn
         // charDataIn receives the stream from the
         // StringBufferInputStream aCommonString
         InputStream charDataIn = new
         StringBufferInputStream(aCommonString);
      }
      catch(IOException e) {
      }
      …
      // perform some process with the stream
   }

   // Where execution begins in a stand-alone executable
   public static void main(String args[]) {
       new Char2Stream(args[0]);
   }
}
```

The stream is piped through the **StringBufferInputStream** class before being sent to the **InputStream** object, **charDataIn**. This technique converts the string of characters into a sequence of bytes so that they can be used in a stream. The stream then can be passed to any of the **InputStreams** to be manipulated later in the program.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y