# Java networking and awt bible


**Introduction**
**Installing the CD-ROM**
**About the Authors**

**Part I—Java Applets And Graphic Components**
**Chapter 1—Applets And Graphics**
**Applets**
**The Essential Applet Methods**
**Applet Parameters**
**Communications Between the Applet and the Browser**
**Using Threads in Applets**
**Inter-Applet Communications Within the Browser**
**Graphics**
**The Graphics Class Concept**
**The Coordinate System of the Drawing Surface**
**Obtaining Graphics Objects**
**The Geometric Primitives**
**The Painting Mode**
**Applet and Associated Class Summaries**
**The Applet and Graphics Project: The Game of Life**
**Project Overview**
**Assembling the Project**
**How It Works**
**Double-Buffered Rendering**
**Overriding Update()**
**Animation Techniques**
**Chapter 2—The Component Class**
**Component Hierarchy**
**Component Positioning**
**Common Component States**
**On-Screen Rendering**
**Delivering and Handling Events**
**Preparing Images for Display**
**Summary of the Component Methods**
**The Component Project: A Hotspot Custom Component**
**Assembling the Project**
**How It Works**

**Part II—Windows And Text Handlers**

## Chapter 3—Toolkit, Window, Container, And Events
A Window in Java
Windows as Pseudo-I/O Devices
Events
Window Repainting in AWT
Components, Containers, and Top-Level Windows
Containers
Windows
Peers and the Toolkit
The Toolkit
Toolkit, Window, Container, and Event Summaries
The Project: FontLab
Assembling the Project
How It Works
## Chapter 4—Windowing Components And Layout Classes
Windowing Components
Layouts
Summary of Windowing Component and Layout Classes
The Layout Demonstration
Assembling the Project
How It Works
## Chapter 5—Handling Text, Dialogs, And Lists
Handling Text
Dialogs in Java
Lists
Text, Dialog, and List Class Summaries
API Reference Interface Application
Building the Project
How It Works

# Part III—Selection And Image Processing Tools
## Chapter 6—Choice, Menus, And Checkboxes
The Choice Class
Menu-Related Classes
Checkbox Classes
Choice, Menu, and Checkbox Summary
The Selections Interface Application
Building Your Application
How It Works
## Chapter 7—Color, Font, Images, And Shapes
Colors
The HSB Color Model
Using Colors
Fonts: The Facts About Rendering Text
Measuring a Font: The FontMetrics Class
Geometric Helper Classes

Graphical Object API Summaries
The Graphical Object Project: Doodle
Assembling the Project
How It Works
Use of Rectangles in Doodle
Chapter 8—AWT Image Processing
Image Data in Java
Passing Image Data: The ImageProducer and ImageConsumer Interfaces
ImageFilters
AWT Image Processing API Summaries
The AWT Image Processing Project: The MultiFilter Application
Assembling the Project
How It Works
*Loading and Storing the Base Image*
*ConfiguredFilterFactory*
*The ContrastFilter*
*The InvertFilter*
*Applying Filters to the Workspace Image*
Adding Your Own ImageFilters to MultiFilter
Chapter 9—AWT Peer Interfaces
AWT Peer Interface Summaries

PART IV—Networking in Java
Chapter 10—Network And Sockets
Client-Server Applications
Connection-Oriented Protocol
Connectionless Protocol
Internet Address
Why Sockets?
Network and Socket Summaries
The Network and Sockets Project: A Client-Server Rendezvous Applet
Building Your Applet
How It Works
Chapter 11—Handling URLs And Networking Exceptions
URLs, Protocols, and MIME
Java and the World Wide Web
URL and Networking Exception Summaries
The URL Class Project
Building the Project
How It Works

Part V—Java Utilities
Chapter 12—Data Structures And Random Number Generation
Dictionary and Hashtable
Vector and Stack
Random Numbers

**Data Structure, Properties, and Random Class Summaries**
**Java Appointment Organizer Applet**
**Building the Project**
**How It Works**
**Chapter 13—Date And Advanced Classes**
**BitSet**
**StringTokenizer**
**Date**
**Observable-Observer**
**Date and Advanced Classes Summaries**
**Appendix A**
**Appendix B**
**Appendix C**
**Appendix D**
**Appendix E**
**Appendix F**
**Index**

# Introduction

Welcome to the world of Java, the language of choice for anyone who wants to develop creative and effective applications on the Internet and the World Wide Web. Java allows programmers to create applets, programs that may be "embedded" inside Web pages; on the other hand, it is a full-fledged object-oriented language that lets you implement object-oriented applications very effectively. The Java Development Kit (JDK) provides a wealth of APIs to help you develop applications quickly. These APIs support developing networking applications, building platform-independent GUIs, multithreading for efficient use of system resources, implementing applets to launch from Web browsers, handling input and output streams, and many others that will help you build applets as well as stand-alone applications. Until now, information on the details of the APIs and how to use them effectively has been hard to come by. We've written the *Java™ Networking and AWT API SuperBible* to give you a head start in exploring this new standard in Web and Internet programming.

## Why a SuperBible?

To make the best use of the Java APIs in developing killer applications, you need nothing less than a SuperBible. The Waite Group's *Java™ Networkingand AWT API SuperBible* is a complete reference to Java's windowing, applet, and networking APIs. It provides more information about Java than any other source. Java's Abstract Window Toolkit (AWT) helps you build graphical user interfaces without having to know the underlying windowing environment. You can develop client-server applications by exploiting Java's networking capabilities even if you don't understand much about the underlying operating system and architecture. You can skip developing often used data structures and utilities over and over again because the JDK provides a set of utilities so you can

concentrate on application design. Java offers so much that it takes a SuperBible to bring it all together for you.

## How Is the Book Organized?

We've divided the *Java™ Networking and AWT API SuperBible* into five logical parts grouped by their functionality. Each chapter discusses a set of classes in a Java package. Under each class, its methods are discussed in alphabetical order. Chapters are arranged by their functionality so you have a detailed walk-through and example of each API. You'll also find a project at the end of each chapter that will give you some hands-on practice using the APIs. Here's a quick rundown of the topics covered in each part and chapter.

### *Part I: Java Applets and Graphic Components*

Part I explains the methods needed to build applets and to use the Graphics class and Component class in windowing applications. You'll find every piece of information about Java applets, graphical components, and generic abstract windowing toolkit components in these chapters.

Chapter 1, *Applets and Graphics,* describes the applet API class and the Graphics class, which are useful for working with graphical components of windowing applications. Chapter 2, *The Component Class,* discusses the Component class needed to develop windowing applications and to subclass many window components for implementation.

### *Part II: Windows and Text Handlers*

Part II, *Windows and Text Handlers,* covers the APIs necessary for creating windows and window components, and for handling text, dialogs, and lists. It also explains the event handlers and APIs that deal with component layout. Chapter 3, *Toolkit, Window, Container, and Events,* presents the APIs used to bind abstract AWT classes to a particular native toolkit implementation and to develop windows and containers for various components. Chapter 4, *Windowing Components and Layout Classes,* discusses the methods to implement windowing components including Button, Canvas, Frame, ScrollBar, Insets, and available Layout APIs. Chapter 5, *Handling Text, Dialogs, and Lists,* covers the Text, Dialog, FileDialog, and List classes. Text handling to display and obtain text data is important for any window program; dialog windows are required to display messages and obtain information from the user.

### *Part III: Selection and Image Processing Tools*

Selection through menus and choice buttons is an important part of a user interface and, in this age of multimedia applications, image processing is vital. Part III describes the functions used for filtering and manipulating images to suit the user's creativity and requirements and the windowing components that enable selection. Chapter 6, *Choice, Menus, and Checkboxes,* covers the Choice, Menu, and Checkbox classes. The classes

that encapsulate color, font, images, and shapes are covered in Chapter 7, *Color, Font, Images, and Shapes.* Chapter 8, *AWT Image Processing,* focuses on the classes and interfaces that help programmers process an image. The details of peers that get created for the AWT components are described in Chapter 9, *AWT Peer Interfaces.*

### *Part IV: Networking in Java*

Part IV, *Networking in Java,* focuses on methods that enable network programming in Java. It covers the functions required to develop client-server applications and to write programs that use information from Web pages across the Internet. Chapter 10, *Network and Sockets,* covers the classes that encapsulate the functionality of Internet addresses and sockets. Chapter 11, *Handling URLs and Networking Exceptions,* describes the Java classes that encapsulate URLs that provide a standard in addressing WWW document pages and exceptions which may be generated in networking applications.

### *Part V: Java Utilities*

The Java utitlity APIs implement utilities and make them available to users. Part V discusses data structure APIs, the random generator class, and advanced utility classes. Chapter 12, *Data Structures and Random Number Generation,* covers the classes Dictionary, Hashtable, Vector, and Stack as well as the Properties class for maintaining the properties of objects. The Enumeration interface that provides the methods necessary to enumerate a given set of elements from the vector or hash table is also covered, along with two additional classes, EmptyStackException and NoSuchElementException, which define the exceptions related to these data structures. Chapter 13, *Date and Advanced Classes,* looks at more advanced utilities. It discusses the class that deals with sets of bits and the class that helps to tokenize a stream of strings, as well as a Wrapper class for finding dates, and advanced classes that encapsulate the Observable-Observer design pattern.

### *Appendices*

Class and interface diagrams, cross references for the exceptions generated, steps involved in native method integration with Java, and information about Java Script are presented as appendices. The class and interface diagrams that present the diagramatic representation of inheritance hierarchy are presented in Appendix A, *Class and Interface Inheritance Diagrams.* For quick reference, Appendix B, *Exception Cross Reference,* presents a list of the exceptions covered in this book. You can integrate native methods written in C/C++ with the Java programs. The steps involved in writing and integrating the native methods are presented in Appendix C, *Writing Native Methods in C/C++.* JavaScript is a scripting lanaguage used to activate the Web and is described in Appendix D, *JavaScript.* Terms used throughout this book are contained in, Appendix E, *Glossary.* Appendix F, *Using the Enclosed CD-ROM,* provides an overview of the contents of the CD that comes with this book.

## Code Your Way to Java Expertise

To help you master Java, each chapter contains many program listings and a complete project. Source code for the complete programs is in the chapters and is also provided on the accompanying CD-ROM, so you can modify and experiment with the code. For ease of use, we've arranged the directories on the CD by chapter number. The CD also has additional information and bonus programs. Enjoy!

### A Matter of Taste

Everybody likes a different cup of java; that is, each person thinks differently and runs Java applications on a distinct platform. In this book, the implementation of programs differs in the way the authors approach a problem, and the appearance of the windows varies depending on the windowing environment the application runs on. These variations are intentional. We believe they will help you understand the behavior of Java in different environments and ultimately make you a more versatile Java applications developer.

We hope you enjoy using this book as much as we've enjoyed writing it. Good luck exploring Java!

# *Part I*
# *Java Applets And Graphic Components*

## Chapter 1
## Applets And Graphics

Packaging interactive content in small, easily distributed objects was a design feature that had high priority to the developers of Java. To meet this design goal, they created the Applet class, along with several objects and interfaces designed to simplify image and audio processing.

An Applet is a custom interface component object, similar in concept to a Windows custom control, or an X-Windows widget. Applet-aware applications (or "applet browsers") can load and construct Applet objects from URLs pointing to .CLASS files anywhere on a network, including the Internet, the largest network of them all. The Java Developer Kit's (JDK) HotJava World Wide Web browser is an example of an applet-aware application. Using it, you can access interactive Applets from anywhere on the Internet, or Applets developed on the local file system. Security features of the Java language ensure distributed applets cannot damage or compromise the security of a local system.

Using the graphical capabilities of Java, applets are visually exciting multimedia elements. Through objects of the class java.awt, Graphics applets can create graphical on-

screen content. The Graphics class is included in this chapter because of the need for applets to display exciting visuals.

Because of all these features, applets have become the preferred method for distributing interactive content on the World Wide Web. A library of reusable, extensible Applets is one of the cornerstones of an Internet content creator's toolkit.

The project for this chapter is an Applet implementation of Conway's *Game of Life*. This project illustrates the use of "double-buffered" animation, the preferred method of graphics animation in Java.

## Applets

Figure 1-1 illustrates the Applet class hierarchy. Most ancestors of Applet in this hierarchy are Abstract Windows Toolkit (AWT) classes. Through them, the Applet class inherits windowing capabilities. Specifically, the Applets display, surface drawing, and mouse and keyboard event handling functionalities are gained through these ancestors. AWT's windowing capabilities are covered in Chapter 9. All examples and discussions in this chapter stop short of utilizing AWT methods other than those that provide applets with their graphical capabilities. But keep in mind the rich set of facilities the AWT classes have when designing your own custom Applet classes.

**Figure 1-1** The Applet class hierarchy

Applet objects are created and controlled by a container application called an *applet browser*. The applet browser (see Figure 1-2) arranges applet objects visually on the screen and dedicates a rectangle of screen space for the applet to display itself. Most applet browsers can manage more than a single Applet object at a time, and actually provide an interface for the Applet instances to communicate with each other.

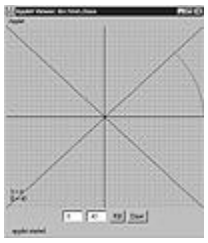

**Figure 1-2** Example of an applet browser (JDK's AppletViewer) with a Java Applet running

### The Essential Applet Methods

The actions of a custom Applet object are ruled by four essential methods: Applet.init, Applet.start, Applet.stop, and Applet.destroy. The browser itself invokes these methods at

specific points during the applet's lifetime. The java.applet.Applet class declares these methods and provides default implementations for them. The default implementations do nothing. Custom applets override one or more of these methods to perform specific tasks during the lifetime of the custom Applet object. Table 1-1 lists these four methods, details when each is called by the browser, and shows what a custom applet's overriding implementation should do.

**Table 1-1** Descriptions of the essential applet methods

| Method | Description |
|---|---|
| init | Called once and only once when the applet is first loaded. Custom implementations allocate resources that will be required throughout the lifetime of the Applet object. |
| destroy | Called once and only once just before the Applet object is to be destroyed. Custom implementations release allocated resources, especially Native resources, which were loaded during init or during the lifetime of the Applet object. |
| start | Called each time the applet is displayed or brought into the user's view on-screen. Custom implementations begin active processing or create processing threads. |
| stop | Called each time the applet is removed from the user's view. Custom implementations end all active processing. Background processing threads should either be destroyed in stop, or put to sleep and destroyed in the destroy method. |

The proper place to allocate objects or load data required by the applet throughout its lifetime is init. This method is called only once during the lifetime of the applet, right after the object is created by the browser. Most custom Applets allocate resources required throughout the lifetime of the Applet object in this method. Another very common operation performed during init is to resize the applet's on-screen display surface using the inherited method Component.resize. Some browsers display applets correctly only if the applet calls resize() in init(). The Component class is described in Chapter 2.

Listing 1-1 defines an applet that plays an audio clip in a continuous loop. This applet could be used to play "theme music" while a particular Web page was being viewed. Notice that the essential methods init, start, stop, and destroy are given overriding implementations to manage what the applet does in the browser.

**Listing 1-1** Example applet

```
import java.Applet.*;

public class ThemeMusicApplet extends Applet {
  AudioClip audclipTheme;

  public void init() {
    // load the audio clip.
    audclipTheme = getAudioClip( getDocumentBase(),
        "images/theme.au" );

    // shrink display surface...never used.
    resize( 0, 0 );
  }

  public void start() {
    // start the audio loop.
    audclipTheme.loop();
  }

  public void stop() {
    // halt the audio loop.
    audclipTheme.stop();
  }

  public void destroy() {
    // release the audio clip from memory.
    audclipTheme = null;
  }
}
```

The browser invokes the ThemeMusicApplet's start method when it is time to present information to the user, and ThemeMusicApplet.start begins playing the audio clip loaded in init. You can see why it is necessary for init to be invoked before any call to start: The audio clip must be loaded before it is played. init is always called before the first invocation of start.

When the applet drops from view, for example because it is scrolled off the screen in the browser or the user opens a different document in the browser, the applet's stop method is called. This is the proper time for a custom Applet to cease any processing. In our example, the continuous audio clip started in ThemeMusicApplet.start is halted in stop.

There are two more important technical notes about start and stop:

- Every call to start has a matching subsequent call to stop.
- The start/stop sequence may be repeated more than one time for the same custom Applet object, for example if the applet is scrolled from the user's view and then scrolled back. When it is scrolled from the user's view, stop will be invoked. When it is scrolled back, start will be invoked for the second time.

When the applet is finally and definitely to be unloaded from memory, destroy() is invoked. This is the appropriate time to delete any resources loaded during init(). The above example applet removes its reference to the AudioClip in destroy. The call to

destroy is guaranteed to occur after the last call to stop. Note that while any resources allocated by an applet will automatically be cleaned up by Java's garbage collection facilities, it is more efficient to remove references to any allocated objects in destroy. Also note that resources allocated by "native" methods will not be cleaned up by the garbage collection facilities. Native resources must be explicitly released in destroy. (Native methods are platform-specific, dynamically loadable libraries accessible from within Java code. For the most part, Applet classes do not use native methods because of the severe security constraints placed on Applet objects. Refer to Appendix C for more information about creating and using native methods in Java.) Figure 1-3 illustrates the sequence of calls to init, start, stop, and destroy by an applet browser.

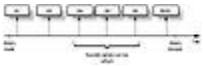

**Figure 1-3** Sequence of init, start, stop, and destroy calls for an Applet

## Applet Parameters

Similar to Java applications, applets can receive and process parameters. Applications receive parameters in the *argv[ ]* argument to the main method. The elements of *argv[ ]* are the command line arguments to the application. Analogous to *argv[ ]*, applet parameters are accessed within the applet code by the Applet.getParameter method.

The use of parameters makes applets extensible. ThemeMusicApplet.init has the theme music relative URL "audio/theme.au" hardcoded into the init method. This listing uses a replacement for init, which makes the ThemeMusicApplet applet play whatever audio file is passed in its "Theme" parameter:

```
public void init() {
  // Get text of the Theme parameter, use as a relative
  // URL.
  String strTheme = getParameter( "Theme" );
  if( null == strTheme ) strTheme = "audio/theme.au";
  audclipTheme = getAudioClip( getDocumentBase(), strTheme );
}
```

Conceptually, the browser maintains an internal listing of all the parameters passed to an embedded Applet object. The getParameter method accesses this internal list and retrieves the values specified for a uniquely named parameter. Our new listing uses the getParameter method to look up the value for the parameter named "Theme". If no such parameter was passed, getParameter would return null.

There is a method defined so that Applet objects can publish a list of valid parameter names, valid values, and a description of each. By overriding the Applet.getParameterInfo method, an Applet class can make this information public to any other object. The default implementation of getParameterInfo simply returns null, but an overriding implementation should return a String[*n*][3] 2-dimensional array where *n* is the number of unique parameters understood by members of the Applet class. Each row of three strings in this array should be of the format:

```
{ "parameter name", "valid value range", "textual description" }
```

There is no strict requirement on the format of any one of these strings. Each one should be suitable for textual display so that someone can read it. For example, the "valid value range" string could be "0-5", meaning the parameters should be an integer between 0 and 5. Listing 1-2 defines a small Applet class called AppletNames that displays, on *System.out*, a listing of all other active Applets currently being managed by the browser and a listing of each Applet object's parameter information retrieved using each Applet object's getParameterInfo method. This Applet class uses its AppletContext to access other active Applet instances. A detailed description of the AppleContext interface and methods follows this discussion of Applet parameters.

**Listing 1-2** Example using the AppletNames class

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

public class AppletNames extends Applet {
    public void init() {
        resize( 0, 0 );
    }

    public void start() {
        Enumeration enumApplets;

        enumApplets = getAppletContext().getApplets();
        while( enumApplets.hasMoreElements() ) {
            Applet appletCurrent =
                    (Applet)enumApplets.nextElement();
            System.out.println( "Next applet:" );
            String[ ][ ] aastrParams =
                    appletCurrent.getParameterInfo();
            boolean fDone = false;
            for( int iIi=0 ; !fDone ; iIi++ ) {
                try {
                    System.out.print( "\tparam: " );
                    System.out.print( aastrParams[iIi][0] );
                    System.out.print( "; value: " );
                    if( appletCurrent.getParameter(
                            aastrParams[iIi][0] ) == null )
                        System.out.println( "<null>" );
                    else
                        System.out.println(
                                appletCurrent.getParameter(
                                        aastrParams[iIi][0] ) );
                } catch( Exception e ) {
                    fDone = true;
                }
            }
        }
```

```
        System.out.println( "End of applet list." );
    }
}
```

Different types of browsers use different methods for passing parameters to applets. For example, applet-aware World Wide Web browsers generally use the HTML <APPLET> container tag to refer to applet code and parameters. Between the <APPLET> and </APPLET> container tags, zero or more <PARAM> tags can appear. These tags have the form <PARAM name=[param-name] value=[param-val]>. No matter how parameters are passed into a particular browser, a loaded applet always uses getParameter to retrieve parameter values.

Listing 1-3 is part of an HTML document with the ThemeMusicApplet embedded in it. The <APPLET> container refers to the ThemeMusicApplet's .CLASS file URL. The <PARAM> tag indicates which audio file to use ("audio/GilligansIsland.au").

**Listing 1-3** HTML document

```
...

<H1>My Theme Music Home Page!</H1>
<P>Just sit right back and you should hear my theme music if
you wait for a moment for audio downloading...

<APPLET SRC="ThemeMusicApplet" width=0 height=0>
<PARAM name="Theme" value="audio/GilligansIsland.au">
</APPLET>

...
```

## Communications Between the Applet and the Browser

Applets obtain information about the state of the browser, what other Applet objects are currently active, what is the current document opened by the browser, and so on, through the java.applet.AppletContext interface. The browser is abstracted by an object implementing this interface.

The browser also exposes some functionalities that an applet can use through this interface. For example, the loading of image and audio files into Java objects is handled transparently through the AppletContext interface. The two overloaded versions of Applet.getAudioClip are actually shallow wrappers around AppletContext.getAudioClip. The full AppletContext interface is detailed in the API descriptions for this chapter.

Between the AppletContext and the Applet is an AppletStub object. Its purpose is to provide a pathway for the exchange of applet-specific data between the AppletContext and the Applet. For example, the parameters for a specific Applet object are accessed by the Applet through AppletStub.getParameter. Applet.getParameter is actually a shallow wrapper around this method. In turn, AppletStub methods are translated into native or custom AppletContext method calls (the implementation of the pathway of data exchange

between the AppletStub and AppletContext is left completely up to the browser developers). An Applet's AppletStub is tightly wrapped by the java.applet.Applet implementation. So much so, that all AppletStub functionalities are exposed as wrapper methods in the java.applet.Applet class. Therefore, a custom applet should never need to use its AppletStub directly. Figure 1-4 illustrates the pathways of data exchange between the Applet, the browser (abstracted by the AppletContext interface), and the AppletStub (the Applet's representative to the browser).

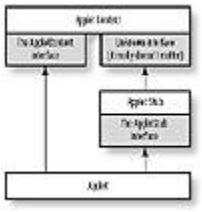

**Figure 1-4** Pathways of data exchange between the Applet, AppletContext, and AppletStub objects

## Using Threads in Applets

Much the same as applications, applets can create Threads to carry on background processing. A typical use of this would be an animation applet. To perform animation, the applet creates a new Thread and starts it running in start. The animation Thread acts as a timer. Every so often, it wakes and draws a new frame in the animation sequence, then suspends itself until the next frame is to be drawn. In the applet's stop method, the animation thread is shutdown. Two versions of this simple animation technique are described in greater detail in the section on the Graphics class and methods. The important point here is that Threads generally are made to begin background processing in an applet's start implementation and either suspended or destroyed in the applet's stop implementation.

You might assume that Threads created by an applet would be automatically halted by the browser when the applet is destroyed, so you wouldn't really need to suspend or destroy a Thread object explicitly in stop. Instead, you could just leave it to Java's garbage collection facilities to destroy your Thread when the Applet object is destroyed. Many browsers, however, do not properly halt secondary applet threads, even after the applet has been destroyed, so the thread continues to execute after the applet has been destroyed. This is a result of applets relying on the Java garbage collection facility to destroy their threads. To ensure your custom applets behave as you want them to, including ceasing when you want them to cease, suspend any secondary threads in Applet.stop, and drop references to them in destroy. As Listing 1-4 shows, the ClockTickApplet demonstrates using this technique in a multithreaded applet.

**Listing 1-4** The ClockTickApplet sourcer

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

```
public class ClockTickApplet extends Applet implements Runnable {
  AudioClip audclipTick;
  AudioClip audclipChime;
  Thread threadTicker;
  Thread threadChimer;

  public void init() {
    // create the new threads, but don't start them
    threadTicker = new Thread( this );
    threadChimer = new Thread( this );

    // get the two clock sounds.
    audclipTick = getAudioClip( getDocumentBase(),
        "audio/tick.au" );
    audclipChime = getAudioClip( getDocumentBase(),
        "audio/chime.au" );

    // just clock sounds, so no display surface is necessary.
    resize( 0, 0 );
  }

  public void destroy() {
    // release threads, which should be suspended by now.
    threadTicker = null;
    threadChime = null;

    // release audio clips, which won't be used any more.
    audclipTick = null;
    audclipChime = null;
  }

  public synchronized void start() {
    // start the ticker thread, which will automatically
    //  start the chimer thread at the correct time.
    threadTicker.start();
  }

  public synchronized void stop() {
    // suspend the ticker, shut down the chimer, which
    //  will be started again by the ticker at the correct
    //  time.
    threadChimer.stop();
    threadTicker.stop();
  }

  // run is what the two threads run in.
  public void run() {
    /* If this is the chime thread, play the chime noise
     *  and quit. */
    if( Thread.currentThread() == threadChimer ) {
      audclipChime.play();
      return;
    }

    /* Sleep for 1 second at a time. Every full minute,
     *  restart the chimer thread. */
```

```
    while( null != threadTicker ) {
      audclipTick.play();
      // GET TIME, IF SECONDS == 0, THEN...
      audclipChimer.start();
      try { sleep( 1000 ); }
        catch( Exception e ) { }
    }
    return;
  }
}
```

### Inter-Applet Communications Within the Browser

You can coordinate the activities of several applets by accessing and manipulating other Applet objects from within Applet code. Using inter-applet communications you could, for instance, have a "gas-gauge" Applet report the status of another Applet, which takes a significant amount of time to initialize.

To obtain references to external Applets from within an applet you use the AppletContext's getApplet and getApplets methods. The AppletNames Applet demonstrates this technique. Once a reference to another Applet is retrieved, your applet code can access any public member variable or method of the external Applet object. This code snippet retrieves an applet named "MyApplet" and calls one of its custom methods.

```
Applet applet = getAppletContext().getApplet( "MyApplet" );
if( ! ( applet instanceof MyAppletClass ) ) return;
MyAppletClass myapplet = (MyAppletClass)applet;

myapplet.CustomFunc();
```

getApplet takes an applet "name" and returns a reference to the associated Applet object. This usage model implies the browser internally stores a unique String name associated with each applet, which can be used to look up the Applet in the internal browser storage.

Similar to defining applet parameters, the method for naming applets depends on the specific browser used. For example, applet-aware World Wide Web browsers generally use a NAME field within the HTML <APPLET> container tag to associate a name with a particular Applet object. This is how an HTML document would embed an applet named "Minnow" of class *ThreeHourTouringBoat*:

```
...
<APPLET src="ThreeHourTouringBoat" name="Minnow" width="100"
height="100">
<PARAM ...>
...
</APPLET>
...
```

## Graphics

Applets are capable of displaying exciting and complex graphics and multimedia visuals. This section explains the specifics of graphical drawing, which is done using objects of the Graphics class and associated classes. Important concepts explaining these classes will be discussed, as well as the most common technique for visual animation, double-buffered graphics.

All graphical drawing operations in Java are performed through objects derived from the Graphics class. Whether you are drawing images downloaded from the Internet, drawing graphical primitives such as rectangles and arcs, or rendering text, all graphical operations are done using a Graphics class instance. Use of the Graphics class is not limited to Applets; it is also used for Java applications that employ graphical elements in windows.

## The Graphics Class Concept

For some beginners, the concept of the Graphics class is a little diffucult to grasp. But it doesn’t have to be. You can think of a Graphics object as analogous to a graphic artist’s drafting table, with its associated drawing tools. It is a station of powerful tools dedicated to creating graphical images.

Each Graphics object is associated with a two-dimensional "drawing surface," analogous to the piece of paper on the drafting table. For example, the drawing surface can be a rectangle of a user’s on-screen desktop, as is the case when dealing with Applets or Windows. Other drawing surface types could also be associated with a Graphics object. The drawing surface could be a binary image, stored in memory and never directly displayed to the user. It could also be a page in a printer, or a fax machine, or even a PostScript or other graphics-format file stored on a disk.

The "tools" of a Graphics object, the methods of the Graphics class, are used to draw onto the associated drawing surface. Rectangles, ovals, arcs, polygons, lines, text, and images can all be drawn onto the drawing surface using the various Graphics class methods.

The internal state of a Graphics object can be described by eight state variables, which can be modified using Graphics class methods.

- The foreground color
- The background color
- The current font
- The painting mode
- The origin of the Graphics object
- The horizontal and vertical scaling factors
- The "clipping" rectangle
- The drawing surface the Graphics object has been associated with

## The Coordinate System of the Drawing Surface

All drawing surfaces use the same two-dimensional coordinate system. The X axis is in the horizontal direction of the drawing surface, and increases from left to right on the drawing surface. The Y axis is in the vertical direction, and increases from top to bottom.

The Graphics object origin defines where its X and Y axes cross, and is identified by the point (0,0). A scaling factor is assigned to both axes, which defines how quickly the coordinates increase along either axis. By default, when the Graphics object is first created, the origin lies in the upper-left corner of the drawing surface, and the scaling factor along both axes is one.

The Graphics object's X and Y axes stretch to what is essentially an infinite distance in all four directions. However, only coordinates within the Graphics object's "clipping rectangle" are of any interest. That's because graphical operations cannot be performed outside this rectangle. Such operations will not result in any sort of error, but neither will they have any effect on the drawing surface.

The clipping rectangle of a Graphics object represents the physical boundaries of the associated drawing surface. For example, a Graphics object associated with a 100 pixel by 100 pixel rectangle of the on-screen desktop will have a clipping rectangle with a width of 100 and a height of 100. For on-screen desktop and in-memory image drawing surfaces, each Graphics coordinate represents a single pixel of the drawing surface. Hence, a 100 pixel by 100 pixel rectangle is represented by a 100 by 100 clipping rectangle in the associated Graphics object.

## Obtaining Graphics Objects

A program cannot create its own Graphics objects, but instead must ask the Java runtime system to create them for specific display surfaces. Without using custom classes implementing native methods, only two types of display surfaces can be accessed through Graphics objects:

- Sections of the on-screen desktop surface are accessed through Graphics objects passed to the update and paint methods.
- In-memory Image objects are accessed through Graphics objects created by Image.createGraphics.

Applets inherit the update and paint methods from the Component class, which the Applet class extends. Both of these methods are called automatically by the Java runtime system when it is time to display information to the user on the desktop. (See Chapter 2, *The Component Class*, for an in-depth discussion of the update and paint methods). This code snippet shows how a custom applet would override the default implementation of paint to control its display surface:

```
public class MyApplet extends Applet {
    ...
```

```
    public void paint(Graphics g) {
        // Draw on the display surface here.
    }

    ...
}
```

A Graphics object is automatically created by the Java runtime system and passed to paint. This Graphics object has a clipping rectangle set to the exact dimensions of the Applet's display surface. (In the cases where only a portion of the Applet must be redrawn, such as when another window temporarily covers part of the Applet's display surface, the dimensions may be smaller.)

The only other method for obtaining a Graphics object is using Image.createGraphics. An applet or application calls this method directly. The Graphics object that is returned is capable of rendering geometric primitives, text, and other Image objects onto the Image. This is useful for the so-called "double-buffered" drawing technique, used widely to effect a smooth transition between animation frames. You'll learn more about this technique in the upcoming discussion of animation.

## The Geometric Primitives

All Graphics objects are able to render several different types of geometric primitive drawing objects on a drawing surface. Table 1-2 lists the various geometric primitives and describes how they are represented by parameters to Graphics rendering methods.

**Table 1-2** Geometric primitives

| Primitive | Representation Through Rendering Methods |
|---|---|
| Rectangle | The point of the upper-left corner of the rectangle relative to the Graphics origin, the rectangle's width and height. |
| Rounded rectangle | The point of the upper-left corner of the rectangle relative to the Graphics origin, the reactangle's width and height. |
| 3D rectangle | The point of the upper-left corner of the rectangle relative to the Graphics origin, the rectangle's width, height, and the raising or depressing implication of the beveled edges. |
| Oval | A bounding rectangle defines the size and shape of the oval. This rectangle is described the same way as a rectangle geometric primitive. |
| Arc | An arc is a section, or pie wedge, of an oval. An arc is described by the bounding rectangle of an oval, the starting angle of the arc, and the angular length of the arc. |

| | |
|---|---|
| Polygon | An ordered set of points defines the vertices of a polygon to Graphics rendering methods. Alternatively, a Polygon object can be used, though Polygons are essentially just an ordered set of vertices. Points are all relative to the Graphics object's origin. |
| Line segment | Two points defining the two endpoints of the line segment. Both points are relative to the Graphics object's origin. |

All primitives can be rendered in either outlined or filled form, except the Line primitive, which cannot be filled. The outlined version of a primitive is rendered using the primitive's "draw" method. For example, Graphics.drawRect will render a rectangle as two sets of parallel lines using the Graphics object's current foreground color. The "fill" method is used to render a filled geometric primitive. Graphics.fillRect will render a solid rectangular block on the display surface using the current foreground color. Listing 1-5 is a complete example of the Nautilus applet, which uses a succession of filled and outlined arc segments (using Graphics.fillArc and Graphics.drawArc) to draw a spiral pattern. The spiral looks something like the cross-section of a nautilus shell, hence the applet class' name.

**Listing 1-5** Example of the Nautilus applet

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

public class Nautilus extends Applet {
    int nAppletHeight, nAppletWidth;
    float flTightness;

    public void init() {
        resize( 500, 500 );
        nAppletHeight = nAppletWidth = 500;

        flTightness =
            (new Float(getParameter("tightness")).floatValue();
    }

    public void paint(Graphics g) {
        int nCenter = nAppletHeight / 2;
        float flRadius = nAppletHeight / 2;
        int nDirection = 1;

        for( int nI=0 ; nI<10 ; nI++ ) {
            int x = ( nAppletWidth / 2 ) - (int)flRadius;
            int y = nCenter - (int)flRadius;

            for( int nJ=0 ; nJ<3; nJ++ ) {
                if( 0 != (nJ + ((1+nDirection) / 2)) % 2 )
```

```
                    g.setColor( Color.red );
                else g.setColor( Color.green );

                g.fillArc( x, y, 2 * (int)flRadius,
                        2 * (int)flRadius,
                        90 + (nDirection * nJ * 60),
                        nDirection * 60 );
            }

            g. setColor( Color.black );
            g.drawArc( x, y, 2 * (int)flRadius,
                    2 * (int)flRadius, 90, nDirection * 180 );

            nCenter += (int)( nDirection * (int)flRadius *
                    ( 1 - flTightness ) );
            flRadius *= flTightness;
            nDirection *= -1;
        }
    }
}
```

The Nautilus applet requires a single parameter, the "tightness" parameter, which describes how tightly the spiral is rendered. The "tightness" parameter's value is a floating point number above 0 and below 1. The closer to 1 this parameter is, the tighter the spiral is. Note that, to preserve code readability, no validation of this parameter has been added. Figure 1-5 is a screenshot of the Nautilus applet run using the JDK's AppletViewer, with a "tightness" parameter of 0.75.

**Figure 1-5** Screenshot of the Nautilus applet

The Nautilus applet illustrates how to use the drawArc and fillArc methods of the Graphics class. Nautilus renders the spiral by drawing successively smaller half-circles made up of alternately colored arc wedges. The code takes advantage of the fact that the sign of the arc length parameter to Graphics.drawArc and Graphics.fillArc defines whether the arc proceeds clockwise or counter-clockwise from the starting angle.

## The Painting Mode

The painting mode of a Graphics object is, by default, set to "overwrite" mode. In this mode, all graphics are rendered by overwriting the pixels of the display surface using the Graphics object's current foreground color. You can force the Graphics object into overwrite mode using Graphics.setPaintMode. When called, this parameterless method places the Graphics into overwrite mode. Expressed pseudo-mathematically, the color of destination pixels after rendering is

```
colorDest(x,y) = graphics.foregroundColor
```

The other method of modifying a Graphics object's painting mode is Graphics.setXORMode. When called, the Graphics object uses XOR mode for rendering geometric primitives, text, or Images on the drawing surface. Three colors are combined mathematically to determine the color of destination pixels after rendering, as follows,

```
colorAfterRendering(x,y) = colorBeforeRendering(x,y)
⊗ graphics.foregroundColor ⊗ graphics.alternateColor
```

where the ⊗symbol represents a bitwise XOR operation. The alternateColor of a Graphics object is specified as the only parameter to Graphics.setXORMode. Listing 1-6 is an example of the Ovals applet, which illustrates XOR painting mode, that can be used to draw contrasting areas of geometric primitives on the drawing surface.

**Listing 1-6** Example of the Ovals applet

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

public class Ovals extends Applet {
    int nAppletWidth;
    int nAppletHeight;

    public void init() {
        resize( 500, 500 );
        nAppletWidth = nAppletHeight = 500;
    }

    public void paint(Graphics g) {
        float flLongAxisLength = nAppletWidth;
        float flShortAxisLength = nAppletHeight / 2;
        boolean fLongAxisVertical = true;
        g.setColor( Color.black );

        for( int nI=0 ; nI<10 ; nI++ ) {
            int x, y, width, height;

            if( fLongAxisVertical ) {
                x = ( nAppletWidth / 2 ) -
                        (int)( flShortAxisLength / 2 );
                y = ( nAppletHeight / 2 ) -
                        (int)( flLongAxisLength / 2 );
                width = (int)flShortAxisLength;
                height = (int)flLongAxisLength;
            } else {
                x = ( nAppletWidth / 2 ) -
                        (int)( flLongAxisLength / 2 );
                y = ( nAppletHeight / 2 ) -
                        (int)( flShortAxisLength / 2 );
                width = (int)flLongAxisLength;
                height = (int)flShortAxisLength;
            }

            g.setXORMode( Color.white );
```

```
            g.fillOval( x, y, width, height );

            g.setPaintMode();
            g.drawOval( x, y, width, height );

            flLongAxisLength *= 0.75;
            flShortAxisLength *= 0.75;
            fLongAxisVertical = ! fLongAxisVertical;
        }

        return;
    }
}
```

Figure 1-6 is a screenshot of Listing 1-6, the Ovals applet, when viewed using the JDK's AppletViewer.

**Figure 1-6** Screenshot of Ovals applet, demonstrating the XOR painting mode of the Graphics class

The Ovals applet uses XOR painting mode to highlight overlapping areas of oval primitives on the drawing surface. The background color of the applet's drawing surface is automatically painted light gray (Color.lightGray or RGB 192, 192, 192) by the Java runtime system.

The foreground color of the Oval's Graphics object is set to Color.black (RGB 0, 0, 0) by default. The XOR alternate color is set to Color.white (RGB 255, 255, 255), the value of the parameter to Graphics.setXORMode. Therefore, the color in which the first oval is rendered can be deduced using the formula presented earlier.

```
colorAfterRendering(x,y) = colorBeforeRendering(x,y)
⊗ graphics.foregroundColor ⊗ graphics.alternateColor
= (RGB 192, 192, 192) ⊗ (RGB 0, 0, 0) ⊗ (RGB 255, 255, 255)
= (RGB 63, 63, 63)
```

When a new oval is rendered so that it intersects an oval that has already been drawn, the intersecting area of these two ovals will be drawn with the color obtained as follows:

```
colorAfterRendering(x,y) = colorBeforeRendering(x,y)
⊗ graphics.foregroundColor ⊗ graphics.backgroundColor
= (RGB 63, 63, 63) ⊗ (RGB 0, 0, 0) ⊗ (RGB 255, 255, 255)
= (RGB 192, 192, 192)
```

In this case the result is Color.lightGray.

It is simple to show that all areas painted with the XOR painting mode using these colors an odd number of times will have the RGB values (63, 63, 63), which is Color.darkGray. All areas painted an even number of times will have the RGB values (192, 192, 192), or Color.lightGray.

## Applet and Associated Class Summaries

Table 1-3 lists the classes and interfaces necessary for developing custom Applet objects in Java. The following sections describe each Class' methods in more detail.

**Table 1-3** Class and interface descriptions

| Class/Interface | Description |
|---|---|
| AppletContext | Exposes services implemented by the applet browser for use by Applet objects. Conceptually, all active Applet objects have access to the same AppletContext. |
| Graphics | Encapsulates a drawing surface, and exposes tools for drawing graphics and rendering text on that drawing surface. A drawing surface may be a rectangle of the desktop, an in-memory image, or even a page in the printer. |
| Applet | Represents an embeddable Applet object. |

## AppletContext

**Purpose**

An interface which abstracts the browser to an Applet. Methods for testing and modifying the current state of the browser are provided as public members of this interface.

**Syntax**

interface AppletContext

**Description**

A running Applet gets its AppletContext using Applet.getAppletContext. Using this interface, the Applet can get and set some parameters of the browser's current state. An Applet can get references to other Applets currently running in the browser, download images and audio clips, and load a new document into the browser through the AppletContext interface.

**PackageName**

*java.applet*

**Imports**

*java.awt.Image, java.awt.Graphics, java.awt.image.ColorModel, java.net.URL, java.util.Enumeration*

**Constructors**

None.

**Parameters**

None.

## getApplet

**Interface**

AppletContext

**Purpose**

Used to facilitate inter-applet communications within a browser.

**Syntax**

public Applet getApplet( String strName );

**Parameters**

None.

***String strName***

This interface method implies the browser stores, with each loaded applet, a unique string to identify that applet. It passes to getApplet one of these unique applet identifiers to gain access to the associated Applet object.

**Description**

Multiple Applet objects can be simultaneously loaded and run by the same browser. Each applet runs within its own Thread. Use this method to access other applets running concurrently. It is completely up to a particular browser how to associate a particular string with an Applet object. For example, most commercial-grade World Wide Web browsers which are applet-aware use the NAME tag in the <APPLET> container tag to associate a name string with a particular applet, as in the HTML snippet below.

```
...
<APPLET CODE=MyApplet.class NAME="Chooser" WIDTH=100 HEIGHT=100>
<PARAM NAME="foo" VALUE="bar">
<PARAM NAME="blepharo" VALUE="spasm">
</APPLET>
```

**Imports**

None.

**Returns**

The Applet object associated with the unique *String strName*. If no applet is associated with *strName*, null is returned or if the applet browser does not provide facilities for inter-applet communications.

**See Also**

The getApplets method of the AppletContext interface

**Example**

The following example tries to find a set of other applets loaded by the browser. A report is written out to *System.out* indicating whether or not each applet could be found. The "applet-list" parameter contains a comma-separated list of applet names to look for. Only up to ten applets will be searched.

```
import java.applet.*;

public class AppletsSearchApplet extends Applet {
  String[ ] astrAppletNames = new String[11];

  public void init() {
    // Nothing displayed on browser, so shrink display surface
    resize( 0, 0 );

    /* Get the "applet-list" parameter, which contains a
     ** comma-separated list of applet names to search for.*/
    String strAppletListParam = getParameter( "applet-list" );
    if( null == strAppletListParam ) {
      System.out.println(
          "No \"applet-list\" parameter provided. );
    return;
    }

    /* Use the applet name list to initialize astrAppletNames.*/
    int iNameIndex = 0;
    int nStartIndex = 0;
    int nNextCommaIndex = 0;

    while( ( -1 != nNextCommaIndex ) && ( iNameIndex < 10 ) ){
      if( -1 == ( nNextCommaIndex =
          strAppletListParam.indexOf( ',', nStartIndex ) ) )
      astrAppletNames[ nNameIndex++ ] =
          strAppletListParam.substring( nStartIndex );
      else
        astrAppletNames[ nNameIndex++ ] =
            strAppletListParam.substring (
                nStartIndex, nNextCommaIndex );

      nStartIndex = nNextCommaIndex + 1;
    }
  }

  public void start() {
    // Look for each named applet in turn and report results.
    for( int iNameIndex = 0
        ; null != astrAppletNames[ iNameIndex ]
        ; iNameIndex++ ) {
      Applet applet = getAppletContext().getApplet(
          astrAppletNames[ iNameIndex ]);
      System.out.print( "Applet " +
          astrAppletNames[ iNameIndex ] + " " );

      if( null == applet )
        System.out.println( "not found!" );
      else
        System.out.println( "found!" );
    }
  }

  // stop() does not need to be implemented.

  public void destroy () {
```

```
    // Release the array of applet name strings.
    astrAppletNames = null;
  }
}
```

## getApplets

**Interface**

AppletContext

**Purpose**

Used to facilitate inter-applet communications within a browser.

**Syntax**

public Enumeration getApplets();

**Parameters**

None.

**Description**

This method allows you to look up all applets currently running in the browser. The browser which implements this method will give you access to all Applet objects currently running in the browser.

**Imports**

None.

**Returns**

An Enumeration object is returned. Each element in the Enumeration is an Applet currently active in the browser. Note that an empty Enumeration, or a return of null, could be interpreted in two ways: Either getApplets() is not fully implemented by the browser, or no other applets are active in the browser. No exact specification currently exists describing what getApplets should return in either of these situations.

**See Also**

The getApplet method of the AppletContext interface

**Example**

The following example uses getApplets and a custom interface, named Namable, to implement an applet-identification facility more complete than the AppletContext facility provided by getApplet and getApplets.

```
interface Namable {
  public String getName();
}
```

Here are the contents of FindNamableApplet.java:

```
public class FindNamableApplet extends Applet implements Namable {
  Hashtable hashAppletsByName;
  String strName;
  String strNameToFind;

  // getName simply returns the "name" parameter value.
  public String getName() {
```

```
    return strName;
  }

  // destroy releases references to object variables.
  public void destroy() {
    hashAppletsByName = null;
    strName = null;
    strNameToFind = null;
  }

  // init fills the applet hash table, and reads in parameters.
  public void init() {
    // Get the two expected parameters.
    if( null == ( strName = getParameter( "name" ) ) ) {
      System.out.println( "Name param missing!" );
      return;
    }
    if( null == ( strNameToFind = getParameter( "find" ) ) ) {
      System.out.println( "Find param missing!" );
      return;
    }

    // Add all Namable applets to hash table.
    Enumeration enumApplets = getAppletContext().getApplets();
    while ( enumApplets.hasMoreElements() ) {
      if( enumApplets.nextElement() instanceof Namable ) {
        Namable applet = (Namable)enumApplets.nextElement();
        hashAppletsByName.put( applet.getName(), applet );
      }
    }
  }

  // start() attempts to find strNameToFind applet, reports results
  public void start() {
    System.out.print( "Applet " + strNameToFind + " " );

    if( hashAppletsByName.containsKey( strNameToFind ) )
      System.out.println( "found." );
    else
      System.out.println( "not found!" );
  }

  // Unnecessary to implement stop().
}
```

## getAudioClip

**Interface**

AppletContext

**Purpose**

Loads an audio file and readies it to be played by the browser.

**Syntax**

public AudioClip getAudioClip( URL url );

**Parameters**

***URL url***

Points to an audio data file to be loaded by the browser.

**Description**

Commercial-grade browsers, especially World Wide Web browsers, have built-in facilities for loading and playing audio files. Applets use the getAudioClip method to load audio files from any URL the browser can understand. Applets should use one of the overloaded Applet.getAudioClip methods to access AudioClips instead of AppletContext.getAudioClip. This method is rarely called by an Applet directly.

**Imports**

*java.net.URL*

**Returns**

The object returned by this function implements the AudioClip interface. If the URL is not understood by the browser, null will be returned or if the browser does not provide this functionality to applets.

**Example**

See the Applet.getAudioClip code example. Applet.getAudioClip is implemented as a simple pass-through to AppletContext.getAudioClip, similar to this code sample:

```
public Applet extends Panel {
  // Other Applet methods declared and implemented...
  // ...

  public AudioClip getAudioClip( URL url ) {
    // Use the AppletContext to load the audio clip
    return getAppletContext().getAudioClip();
  }
}
```

## getImage

**Interface**

AppletContext

**Purpose**

To load an image from a URL and prepare it for rendering on a display surface.

**Syntax**

public Image getImage( URL url );

**Parameters**

***URL url***

Points to an image file to be loaded by the browser.

**Description**

Java applications must implement methods for reading and interpreting image files, and converting the image data into Image objects. Applets may have this functionality exposed to them by the browser through the AppletContext.getImage method. Browsers that can load and interpret various image formats, such as GIF, JPEG or TIFF, can provide that capability to

applets. Applets simply provide a URL pointing to an image file in a recognized format. No methods are provided for an applet to query which image formats are supported by a browser. Therefore, it is usually a good idea to only try to load images in very common graphics formats, such as GIF or JPEG.

**Imports**

*java.awt.Image*

**Returns**

An Image object will be returned by this object, or null if this facility is not supported by the browser. The reaction of this method when the URL refers to an unsupported protocol, or when the image file format is unrecognized, is unspecified. Generally, it can be assumed that null will be returned if this capability it not provided by the browser.

**See Also**

The Image class

**Example**

The following sample applet loads and displays an image. A relative URL to the image to be loaded is passed to the applet as the "image" parameter. That parameter will be interpreted relative to the URL returned by Applet.getDocumentBase. An object of the MyImageObserver class is needed to receive an error message if there is a problem with loading or displaying the image.

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

class MyImageObserver implements ImageObserver {
  public boolean imageUpdate( Image image, int nInfoFlags,
      int nX, int nY, int nCx, int nCy ) {
    // If an error has been detected, report it.
    if( nInfoFlags & ImageObserver.ERROR )
    System.out.println( "Error with the image." );
    return true;
  }
}

public class ImageApplet extends Applet {
  Image image;
  MyImageObserver mio = new MyImageObserver;

  public void init() {
    // resize to some fixed size: large images will be clipped
    resize( 100, 100 );

    // Get the relative URL for the image
    String strRelativeURL = getParameter( "image" );
    if( null == strRelativeURL ) {
      System.out.println( "Image parameter missing." );
      return;
    }
    // Load the image.
    image = getAppletContext().getImage(
        getDocumentBase(), strRelativeURL );
    if( null == image )
```

```
      System.out.println( "Unable to load image." );
  }

  public void destroy() {
    // Get rid of reference to image and image observer.
    image = null;
    mio = null;
  }

  public void paint( Graphics g ) {
    // Paint image on display surface, if image exists
    if( null != image )
      g.drawImage( image, 0, 0, mio );
  }
}
```

## showDocument

**Interface**

AppletContext

**Purpose**

Opens a new document in the browser. An overloaded version exists to specify the name of the target browser frame.

**Syntax**

public void showDocument( URL url ); public void showDocument( URL url, String target );

**Parameters**

***URL url***

Points to the document to be opened by the browser. If the protocol referred to by the URL is not recognized by the browser, this call will be ignored. If the document format implied by the URL's file name is not recognized by the browser, this call will be ignored.

***String target***

You may specify a named browser-frame for the new document to appear in. Table 1-4 lists the valid values for this parameter

**Table 1-4** Valid values of the String target parameter

| Value | Meaning |
|---|---|
| "_self" | The same frame the Applet is embedded in. |
| "_parent" | The parent frame of the frame the Applet is embedded in. |
| "_top" | The top-level window this Applet appears in. |
| "_blank" | A new, top-level unnamed frame. |
| <*other*> | Any other name causes the browser to search for an extant frame with this name. If none exists, then a new top-level frame with this name is created. |

**Description**

In the abstract, Applets are seen as being embedded in distributed "documents," such as World Wide Web pages. When implemented, this method allows the applet to force the browser to open a particular document pointed to by a URL. Like all other methods in this interface, a particular browser may not implement this method, in which case the browser will simply ignore a call to this method. If the second overloaded version of this method is used, then the document will be opened in a browser frame with the same name as the target parameter.

**Imports**

*java.net.URL*

**Returns**

The Applet object associated with the unique String strName. If no applet is associated with strName, null is returned or if the applet browser does not provide facilities for inter-applet communications.

**See Also**

The getDocumentBase method of the Applet class

**Example**

This applet asks the browser to reload the current document whenever the Applet's stop method is invoked. (Not generally a very nice thing to do.)

```
public class RestartingApplet extends Applet {

    public void stop() {
        AppletContext ac = getAppletContext();
        if(null != ac)
            ac.showDocument(getDocumentBase());
    }

}
```

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home



## Graphics

**Purpose**

An AWT Component (such as an Applet) uses a Graphics object to draw on a display surface.

**Syntax**

public class Graphics

**Description**

A Graphics object is always associated with a "display surface." The display surface can be a rectangle of the on-screen desktop, an Image in memory, or potentially any rectangular area that can be drawn on. You use the Graphics class methods to render graphics and text on the display surface assoicated with the Graphics object. Figure 1-7 shows the class diagram for the Graphics class.

**PackageName**

*java.awt*

**Imports**

*java.awt.*, java.image.ImageObserver*

**Constructors**

None. You cannot create Graphics objects directly, but instead get them from other Java API methods. For example, Image.getGraphics returns a Graphics object, which uses the Image as its drawing surface.

**Parameters**

None.

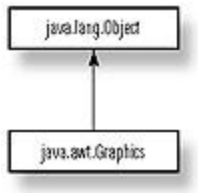


**Figure 1-7** Class diagram for the Graphics class

## clearRect

**ClassName**

Graphics

**Purpose**

To erase the specified rectangle using the background color of the display surface associated with the Graphics object.

**Syntax**

public abstract void clearRect(int x, int y, int width, int height);

**Parameters**

***int x, int y, int width, int height***

These four parameters define the rectangle to be erased on the display surface.

**Description**

This method is used to erase a rectangle from the display surface. The associated display surface's background color is used to fill the specified rectangle. This is a legacy method which was never removed from the alpha release of Java. Use of this method is not advised. Instead, use Graphics.fillRect, specifying the color you want to use to erase the rectangle. It is an unfortunate but true fact that the Java API does not specify an overloaded version of this method which takes a Rect object as a parameter. The origin and extent of the rectangle must be explicitly provided in the four parameters to this method.

**Imports**

None.

**Returns**

None.

**See Also**

The fillRect method of the Graphics class

**Example**

Use of this method is not advised, so an example is not provided.

## clipRect

**ClassName**

Graphics

**Purpose**

To shrink the clipping rectangle of the Graphics object.

**Syntax**

public abstract void clipRect(int x, int y, int width, int height);

**Parameters**

***int x, int y***

These four parameters define the new clipping rectangle for the Graphics object.

***int width***

***int height***

**Description**

Use this method to modify the clipping rectangle of the Graphics object. The clipping rectangle restricts drawing on the drawing surface to within the rect. The resultant clipping rectangle is the intersection of the current clipping rect and the new one defined by the parameters passed to this method. That is, the clipping rectangle can never be made larger, only smaller. This is to prevent ill-behaved Components from enlarging their clipping rectangles to include the entire desktop and then drawing all over the desktop. The clipping rectangle is one of the internal state variables of all Graphics objects.

**Imports**

None.

**Returns**

None.

**See Also**

The create method of the Graphics class

**Example**

```
public class MyApplet extends Applet {
    public void paint(Graphics g) {
        // Get the current clipping rectangle
        Rect rectClip = g.getClipRect();

        // Try to enlarge the clipping rectangle. Since
        // you can't enlarge clipping rectangles, resultant
        // clipping rect will be same size as original.
        g.clipRect(0, 0, 1000, 1000);

        // Shrink clipping rect by 10 on all sides. This
        // always returns intersection of old and new
        // rects.
```

```
        g.clipRect(rectClip.x+10, rectClip.y+10,
                rectClip.width-20, rectClip.height-20);
    }
}
```

## copyArea

**ClassName**

Graphics

**Purpose**

Copies a rectangle of the display surface to a new location on the display surface.

**Syntax**

public abstract void copyArea(int x, int y, int width, int height, int dx, int dy);

**Parameters**

***int x, int y, int width, int height***

These four parameters define the source rectangle to copy from.

***int dx***

The origin of the target rectangle.

***int dy***

**Description**

Copies the pixels of a rectangle of the display surface to another rectangle of the same display surface. Note that the source rectangle may reside outside the clipping rectangle of the Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The drawImage method of the Graphics class

**Example**

This example clears the target area by using the XOR drawing mode of the Graphics object. The target rectangle will be painted with the alternate XOR color after the operation is complete.

```
public void paint(Graphics g) {
    g.setXORMode(Color.white);
    Rect rectClip = g.getClipRect();

    // Copy clipping rectangle over itself, which causes
    // image to disappear, leaving only the alternate
    // color (Color.white, in this case).
    g.copyArea( clipRect.x, clipRect.y, clipRect.width,
            clipRect.height, clipRect.x, clipRect.y);
}
```

## create

**ClassName**

Graphics

**Purpose**

Creates a copy of this Graphics object.

**Syntax**

public abstract Graphics create();
public abstract Graphics create(int x, int y, int width, int height );

**Parameters**

***int x, int y, int width, int height***

These parameters define the display surface of the returned Graphics object. The resultant clipping rectangle will be equal to the intersection of the original Graphics object's clipping rectangle and the rectangle defined by these parameters.

**Description**

Creates a clone of the original Graphics object, attached to the same display surface and with the same internal state as the original Graphics object. The second overloaded version makes a new Graphics object attached to a specific rectangle of the original Graphics object's display surface. As the example below illustrates, the create method is most useful when you want to shrink the clipping rectangle, but get the original, larger clipping rectangle back later.

**Imports**

None.

**Returns**

A Graphics object is returned which is a clone of the original. If the second overloaded version of this method is used, then the clipping rectangle of the resultant Graphics object will be equal to the intersection of the clipping rectangle of the original Graphics and the rectangle defined by the parameters to this method. Also, the origin of the resultant Graphics object will be at the point defined by the *x* and *y* parameters to this method.

**See Also**

The clipRect method of the Graphics class

**Example**

This example uses the create method to temporarily shrink the clipping rectangle of a Graphics object.

```
public void paint(Graphics g) {
    // Get clipping rect of original Graphics
    Rect rectClip = g.getClipRect();

    // create Graphics with smaller clipping rect.
    Graphics gTemp = g.create(rectClip.x+10, rectClip.y+10,
            rectClip.width-20, rectClip.height-20);

    // use smaller clip rect in temp Graphics
    ...

    // Clip rect of original Graphics is still preserved
    // (can't do that with Graphics.clipRect!).
}
```

## translate

**ClassName**

Graphics

**Purpose**

Moves the origin of the Graphics' coordinate system.

**Syntax**

public abstract void translate(int x, int y);

**Parameters**

***int x, int y***

These two parameters define a point which is the new origin of the display surface. The parameters are offsets from the original Graphics object's origin.

**Description**

Modifies the origin of the Graphics object. The origin is one of the internal state variables of Graphics objects.

**Imports**

None.

**Returns**

None.

**Example**

```
public void paint(Graphics g) {
    // Move the origin ten points to the right and down
    // before using the Graphics object...
    g.translate(10, 10);

    ...
}
```

## draw3DRect

**ClassName**

Graphics

**Purpose**

Renders a raised or depressed rectangle on the Graphics' display surface.

**Syntax**

public void draw3DRect(int x, int y, int width, int height, boolean raised);

**Parameters**

***int x, int y, int width, int height***

The dimensions of the rectangle to be rendered on the display surface

***boolean raised***

This parameter tells whether the beveling should imply a raised or depressed effect for the 3D rectangle.

**Description**

Renders a rectangle with beveled edges to create a 3D visual effect. The beveling can either imply a raised or depressed 3D rectangle. The shades of the beveling are choosen based on the current drawing color of the Graphics object. The darker shading will be roughly 70 percent as bright as the current drawing color. The lighter color will be roughly 140 percent as bright as the current drawing color. The beveling will be exactly one pixel wide.

**Imports**

None.

**Returns**

None.

**See Also**

The fill3DRect and drawRect methods of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Draw a raised 3D rectangle, 20x20 in size
    g.draw3DRect(0, 0, 20, 20, true);

    // Draw a depressed 3D rectangle of same dimensions
    g.draw3DRect(10, 10, 20, 20, false);
}
```

## drawArc

**ClassName**

Graphics

**Purpose**

Renders the arc of an oval's wedge on the Graphics' display surface.

**Syntax**

public abstract drawArc(int x, int y, int width, int height, int startAngle, int arcAngle);

**Parameters**

***int x, int y, int width, int height***

The dimensions of the rectangle bounding an oval. The arc is taken as part of the circumference of this oval.

***int startAngle***

Measured in degrees, this defines the start of the arc. Zero degrees lies in the "3 o'clock" position.

***int arcAngle***

Measured in degrees, the distance of the arc around the oval. A positive value indicates a counter-clockwise direction around the oval. Negative indicates clockwise.

**Description**

Renders an arc of an oval on the display surface. The oval is defined by a bounding rectangle, and the arc is described by a starting and stopping angle in degrees.

**Imports**

None.

**Returns**

None.

**See Also**

The fillArc method of the Graphics class

**Example**

See the Nautilus example applet earlier in the chapter.

## drawBytes

**ClassName**

Graphics

**Purpose**

Renders an array (or subarray) of bytes that are interpreted as ASCII character values, on the Graphics' display surface.

**Syntax**

public abstract void drawBytes(byte data[ ], int offset, int length, int x, int y);

**Parameters**

***byte data[ ]***

The array of byte data of ASCII characters to render on the display surface.

***int offset***

The zero-based index of the first character to render.

***int length***

The number of ASCII characters to render.

***int x***

The horizontal offset from the origin to render the characters on the drawing surface.

***int y***

The vertical offset of the baseline where the text is to be rendered. This is measured from the current origin of the Graphics context.

**Description**

drawBytes renders text on the drawing surface taken from a subarray of an array of bytes. The bytes are interpreted as 8-bit ASCII character values. The current font and drawing color of the Graphics is used to render the text.

**Imports**

None.

**Returns**

None.

**See Also**

The drawString, drawChars, and setFont methods of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Initialize an array of bytes with ASCII character
    // values
    byte[ ] ab = new byte[10];
    ab[0] = 'G';
    ab[1] = 'i';
    ab[2] = 'l';
    ab[3] = 'l';
    ab[4] = 'i';
    ab[5] = 'g';
    ab[6] = 'a';
    ab[7] = 'n';

    // Render the ASCII characters to the drawing surface.
    // Baseline is 20 pixels below the origin.
    g.drawBytes(ab, 0, 8, 0, 20);
```

```
}
```

## drawChars

**ClassName**

Graphics

**Purpose**

Renders an array of ASCII characters on the drawing surface. The array can be a subarray of a larger set of characters.

**Syntax**

public abstract void drawChars(char data[ ], int offset, int length, int x, int y);

**Parameters**

***char data[ ]***

The array of ASCII characters to render on the display surface.

***int offset***

The zero-based index of the first character to render.

***int length***

The number of ASCII characters to render.

***int x***

The horizontal offset from the origin to render the characters on the drawing surface.

***int y***

The vertical offset of the baseline where the text is to be rendered. This is measured from the current origin of the Graphics context.

**Description**

drawChars renders text on the drawing surface taken from a subarray of an array of characters. The array values are interpreted as ASCII character values. The Graphics object's current font and drawing color are used to render the characters on the Graphics' display surface.

**Imports**

None.

**Returns**

None.

**See Also**

The drawBytes, drawString, and setFont methods of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Initialize an array of chars with ASCII character
    // values
    char[ ] ac = new char[10];
    ac[0] = 'S';
    ac[1] = 'k';
    ac[2] = 'i';
    ac[3] = 'p';
    ac[4] = 'p';
    ac[5] = 'e';
    ac[6] = 'r';

    // Render the ASCII characters to the drawing surface.
```

```
    // Baseline is 20 pixels below the origin.
    g.drawChars(ac, 0, 8, 0, 20);
}
```

## drawImage

**ClassName**

Graphics

**Purpose**

Renders an Image on the Graphics object's display surface.

**Syntax**

public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer);

public abstract boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer);

**Parameters**

***Image img***

The Image object to be displayed.

***int x, int y***

The coordinate of the upper-left corner of the image on the drawing surface

***int width, int height***

Using the second overloaded version of this method, you can specify the size of the target rectangle to copy the Image to. By using a different size than the original Image object, you can stretch/shrink the Image on the drawing surface.

***ImageObserver observe***

Notifies whether the image is complete or not. (See comments)

**Description**

The passed Image is copied to the drawing surface. The second overloaded version of this method allows you to stretch/shrink the Image on the drawing surface. The ImageObserver is notified about progress of copying the image to the drawing surface. This is useful especially if the Image object is created from a URL pointing to a .GIF or other graphics-format on the network. If, for example, the URL does not actually point to an image, or to an incomplete image file, the ImageObserver object is notified. This results in a little more overhead in coding, but the increase in coding results in more robust applets and applications. Note that all Components, including all Applets, automatically implement the ImageObserver interface. The default implementation of this interface causes the Component to be repainted whenever an update of the image is read in.

**Imports**

None.

**Returns**

None.

**See Also**

The getImage method of the Applet class; the getImage method of the Toolkit class; the ImageObserver interface; and the MediaTracker class

**Example**

This applet creates an image from a URL in its init implementation. In paint, that image is rendered twice on the applet's display surface, once at the Image's default size, and a second time stretched to fit the entire surface of the Applet.

```
public class MyApplet extends Applet {
    Image _img = null;

    // Create the Image from a URL in init.
    public void init() {
        _img = getImage(
            new URL("http://www.sample.com/sample.img"));
    }

    // In paint, render the image once stretched and once not
    // stretched.
    public void paint(Graphics g) {
        // Make sure _img is not null.
        if(null == _img)
            return;

        // Render the imge stretched, using this Applet
        // as the ImageObserver.
        g.drawImage(_img, 0, 0, size().width,
                size().height, this);

        // Render the image not stretched.
        // Again, using this Applet as the ImageObserver.
        g.drawImage(0, 0, this);
}
```

## drawLine

**ClassName**

Graphics

**Purpose**

Renders a line between two points on the Graphics object's display surface.

**Syntax**

public abstract void drawLine(int x1, int y1, int x2, int y2);

**Parameters**

***int x1***

One endpoint of the line segment to render on the drawing surface.

***int y1***

***int x2***

Other endpoint of the line segment to render on the drawing surface.

***int y2***

**Description**

Renders a line between the two points on the drawing surface. The current drawing color is used to render the line. Note that there is no way, using the Java API, to specify lines with a width greater than one pixel. To achieve wide lines, you must render multiple side-by-side lines on the display surface.

**Imports**

None.

**Returns**

None.

**Example**

The example uses the current drawing color to render a 5x5 grid on the Graphics' clipping rectangle.

```
public void paint(Graphics g) {
    // Get the clipping rectangle and figure out grid
    // cell width and height from it.
    Rect rectClip = g.getClipRect();
    int cxCellWidth = rectClip.width / 5;
    int cyCellHeight = rectClip.height / 5;

    // Draw the grid.
    for( int ii=0 ; ii<=5 ; ii++ )
        for( int jj=0 ; jj<=5 ; jj++ ) {
            g.drawLine(ii*cxCellWidth, 0, ii*cxCellWidth,
                    size().height);
            g.drawLine(0, jj*cyCellHeight, size().width,
                    jj*cyCellHeight);
        }
}
```

## drawOval

**ClassName**

Graphics

**Purpose**

Renders an oval defined by a bounding rectangle.

**Syntax**

public abstract void drawOval(int x, int y, int width, int height);

**Parameters**

***int x, int y, int width, int height***

These parameters define the bounding rectangle of the oval.

**Description**

Renders an oval on the display surface. The oval is defined as being bound by the four sides of the rectangle described by the input parameters to this method. The color of the resulting oval is the current drawing color of the Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The fillOval method of the Graphics class

**Example**

See the Ovals example applet earlier in this chapter.

## drawPolygon

**ClassName**

Graphics

**Purpose**

Renders a polygon on the Graphics object's display surface.

**Syntax**

public abstract void drawPolygon(int xPoints[ ], int yPoints[ ], int nPoints);
public void drawPolygon(Polygon p);

**Parameters**

***int xPoints[ ], int yPoints[ ]***

These two arrays describe an ordered set of points which define the vertices of a polygon to be rendered on the drawing surface.

***int nPoints***

The number of vertices of the polygon to be rendered. This is also the number of elements in both the xPoints[ ] and yPoints[ ] arrays.

***Polygon p***

A Polygon object which describes the polygon to be rendered on the drawing surface

**Description**

Two overloaded versions of this method, both allow you to render a multisided polygon on the drawing surface. The logic of the rendering algorithm automatically closes the polygon by connecting the last point of the polygon to the first.

**Imports**

None.

**Returns**

None.

**See Also**

The fillPolygon method of the Graphics class; the Polygon class

**Example**

```
public void paint(Graphics g) {
    // Instantiate two arrays of coordinates with three
    // values each, indicating the vertices of a triangle.
    int[ ] acx = new int[3];
    int[ ] acy = new int[3];

    acx[0] = 0 ; acy[0] = 0 ;  // the point (0 , 0 )
    acx[1] = 10; acy[1] = 0 ;  // the point (10, 0 )
    acx[2] = 10; acy[2] = 10;  // the point (10, 10)

    // Draw the polygon represented by these three points
    g.drawPolygon(acx, acy, 3);
}
```

## drawRect

**ClassName**

Graphics

**Purpose**

Renders a simple rectangle on the drawing surface.

**Syntax**

public abstract void drawRect(int x, int y, int width, int height);

**Parameters**

***int x, int y, int width, int height***

These parameters define the rectangle to be rendered on the drawing surface.

**Description**

Renders the rectangle defined by the four parameters on the Graphics' display surface. Use drawRect(rect.x, rect.y, rect.width-1, rect.height-2) to render the outline of a particular Rect object. The rectangle is rendered using the Graphics' current drawing color.

**Imports**

None.

**Returns**

None.

**See Also**

The fillRect method of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Draw a 10x10 rectangle in the upper-left corner of
    // drawing surface.
    g.drawRect(0, 0, 10, 10);
}
```

## drawRoundRect

**ClassName**

Graphics

**Purpose**

Renders a rectangle with rounded corners on the Graphics' display surface.

**Syntax**

public abstract void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight);

**Parameters**

***int x, int y, int width, int height***

These parameters define the rectangle to be rendered on the drawing surface.

***int arcWidth, int arcHeight***

These two parameters define the width and height of the arcs that are each of the rounded corners. These number are both interpreted as the diameters of the arc at the four corners. So the width and height of the arc at each corner are 1/2 of the arcWidth and arcHeight parameters, respectively.

**Description**

Renders a rectangle with rounded corners on the drawing surface. Through the parameters to this method, both the width and height of the corner arcs can be defined. The outline of the rectangle is rendered using the current drawing color of the Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The fillRoundRect method of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Draw a rounded rect around the circumference of this
    // Component. Make the corners 5 pixels wide and tall
    // at the arc.
    Dimension d = size();
    drawRoundRect(0, 0, d.width-1, d.height-1, 10, 10);
}
```

## drawString

**ClassName**

Graphics

**Purpose**

Renders a string of text on a drawing surface using the Graphics' current font and drawing color.

**Syntax**

public abstract void drawString(String str, int x, int y);

**Parameters**

***String str***

String containing the text to be rendered on the drawing surface. The entire string will be rendered. To render a substring of a String object, use either the drawBytes or drawChars Graphics methods.

***int x***

The horizontal offset from the origin to render the String on the drawing surface.

***int y***

The vertical offset of the baseline where the text is to be rendered. This is measured from the current origin of the Graphics context.

**Description**

Draws the full String using the current font and drawing color. The left-most point of the baseline is indicated by the *x* and *y* parameters.

**Imports**

None.

**Returns**

None.

**See Also**

The drawBytes, drawChars, and setFont methods of the Graphics class

## fill3DRect

**ClassName**

Graphics

**Purpose**

Renders a filled, raised, or depressed rectangle on the Graphics' display surface.

**Syntax**

public void fill3DRect(int x, int y, int width, int height, boolean raised);

**Parameters**

***int x, int y, int width, int height***

These parameters define the dimensions of the rectangle to draw on the display surface.

***boolean raised***

This parameter tells whether the beveling should imply a raised or depressed effect for the 3D rectangle.

**Description**

Renders a rectangle with beveled edges to create a 3D visual effect. The beveling can either imply a raised or depressed 3D rectangle. The shades of the beveling are choosen based on the current drawing color of the Graphics object. The darker shading will be roughly 70 percent as bright as the current drawing color. The lighter color will be roughly 140 percent as bright as the current drawing color. The beveling will be exactly one pixel wide. The inside of the rectangle will be filled using the current drawing color of the Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The draw3DRect and fillRect methods of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Draw a filled raised 3D rectangle, 20x20 in size
    g.fill3DRect(0, 0, 20, 20, true);
    // Fill a depressed 3D rectangle of same dimensions
    g.fill3DRect(10, 10, 20, 20, false);
}
```

## fillArc

**ClassName**

Graphics

**Purpose**

Renders a wedge of an oval on the Graphics' display surface.

**Syntax**

public abstract fillArc(int x, int y, int width, int height, int startAngle, int arcAngle);

**Parameters**

***int x,int width, int height***

The dimensions of the rectangle bounding an oval. The arc is a wedge of *int y* this oval.

***int startAngle***

Measured in degrees, this defines the start of the arc. 0 degrees lies in the "3 o'clock" position.

***int arcAngle***

Measured in degrees, the distance of the arc around the oval. A positive value indicates a counter-clockwise direction around the oval. Negative value indicates a clockwise direction.

**Description**

Draws a wedge of an oval. The oval is defined by a bounding rectangle, and the wedge is described by a starting and stopping angle in degrees.

**Imports**

None.

**Returns**

None.

**See Also**

The drawArc method of the Graphics class

**Example**

See the Nautilus example Applet earlier in the chapter.

Previous | Table of Contents | Next

Products | Contact Us | About Us | Privacy | Ad Info | Home



## fillOval

**ClassName**

Graphics

**Purpose**

Renders a filled oval defined by a bounding rectangle.

**Syntax**

public abstract void fillOval(int x, int y, int width, int height);

**Parameters**

***int x, int y, int width***

These parameters define the bounding rectangle of the oval to be rendered.

***int height***

**Description**

Renders a filled oval on the display surface. The oval is defined as being bound by the four sides of the rectangle described by the input parameters to this method. The color of the resulting oval is the current drawing color of the Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The drawOval method of the Graphics class

**Example**

See the Ovals example Applet earlier in this chapter.

## fillPolygon

**ClassName**

Graphics

**Purpose**

Renders a filled polygon on the Graphics object's display surface.

**Syntax**

public abstract void fillPolygon(int xPoints[ ], int yPoints[ ], int nPoints); public void fillPolygon(Polygon p);

**Parameters**

***int xPoints[ ], int nPoints, int yPoints[ ]***

These two arrays describe an ordered set of points which define the vertices of a polygon to render on the drawing surface.

The number of vertices of the polygon to be rendered. This is also the number of elements in both the xPoints[ ] and yPoints[ ] arrays.

***Polygon p***

A Polygon object which describes the polygon to render on the drawing surface.

**Description**

Two overloaded versions of this method both allow you to render a multisided polygon on the drawing surface. The logic of the rendering algorithm automatically closes the polygon by connecting the last point of the polygon to the first. The odd-even filling algorithm is used to fill polygons. So for complex polygons, internal areas may or may not get filled. The general rule of thumb is that an area will be filled if a line segment drawn from outside the polygon to the area within the polygon crosses an odd number of the polygon's line segments. If an even number is crossed, then the area will not be filled. For example, the center of a pentagram would not get filled, while each of the five arms of the pentagram would get filled.

**Imports**

None.

**Returns**

None.

**See Also**

The drawPolygon method of the Graphics class, the Polygon class

**Example**

```
public void paint(Graphics g) {
    // Instantiate two arrays of coordinates with three
    // values each, indicating the vertices of a triangle.
    int[ ] acx = new int[3];
    int[ ] acy = new int[3];

    acx[0] = 0 ; acy[0] = 0 ;  // the point (0 , 0 )
    acx[1] = 10; acy[1] = 0 ;  // the point (10, 0 )
```

```
    acx[2] = 10; acy[2] = 10;  // the point (10, 10)

    // Draw the polygon represented by these three points
    g.fillPolygon(acx, acy, 3);
}
```

## fillRect

**ClassName**

Graphics

**Purpose**

Renders a simple filled rectangle on the drawing surface.

**Syntax**

public abstract void fillRect(int x, int y, int width, int height);

**Parameters**

***int x, int y, int width, int height***

These parameters define the rectangle to render on the drawing surface.

**Description**

Renders the filled rectangle described by the four parameters on the Graphics' display surface. Use drawRect(rect.x, rect.y, rect.width-1, rect.height-2) to render a particular Rect object. The rectangle is rendered using the Graphics' current drawing color.

**Imports**

None.

**Returns**

None.

**See Also**

The drawRect method of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Draw a 10x10 rectangle in the upper-left corner of
    // drawing surface.
    g.fillRect(0, 0, 10, 10);
}
```

## fillRoundRect

**ClassName**

Graphics

**Purpose**

Renders a filled rectangle with rounded corners on the Graphics' display surface.

**Syntax**

public abstract void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight);

**Parameters**

***int x, int y, int width, int height***

These parameters define the rectangle to render on the drawing surface.

***int arcWidth, int arcHeight***

These two parameters define the width and height of the arcs that are each of the rounded corners. These numbers are both interpreted as the diameters of the arc at the four corners. So the width and height of the arc at each corner is 1/2 of the arcWidth and arcHeight parameters, respectively.

**Description**

Renders a filled rectangle with rounded corners on the drawing surface. Through the parameters to this method, both the width and height of the corner arcs can be defined. The rectangle is rendered using the current drawing color of the Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The drawRoundRect method of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Draw a rounded rect around the circumference of this
    // Component. Make the corners 5 pixels wide and tall
    // at the arc.
    Dimension d = size();
    g.fillRoundRect(0, 0, d.width-1, d.height-1, 10, 10);
}
```

## getClipRect

**ClassName**

Graphics

**Purpose**

Returns a Rect whose members are set to the origin and dimensions of the current clipping rectangle for this Graphics object.

**Syntax**

public abstract Rectangle getClipRect();

**Parameters**

None.

**Description**

Allows you access to the clipping rectangle dimensions and placement, which is one of the internal state variables of the Graphics object. Drawing operations are only valid within the clipping rectangle. Graphics-intensive applet or applications are easily optimized in the paint method by performing only those operations valid within the clipping rectangle. For example, an application which renders lots of images to the drawing surface would see vast improvements in speed by only drawing those images which overlap with the clipping rectangle, since any drawing outside the clipping rectangle is ignored anyway.

**Imports**

None.

**Returns**

A Rectangle object which represents the position and size of the clipping rectangle relative to the origin of the Graphics object.

**See Also**

The clipRect and create methods of the Graphics class

**Example**

This paint method is responsible for rendering a 100x100 grid of Images on the drawing surface. The method is optimized by only rendering those Images which fall within the Graphics' clipping rectangle.

```
// Assume a 100x100 array of Images has been initialized...
// each image is 10x10 pixels in size.
Image[ ][ ] _aaimg = new Image[100][100];
public void paint(Graphics g) {
    // get the clipping rectangle
    Rect rectClip = g.getClipRect();

    // only draw those images which overlap with the
    // clipping rectangle.
    int ii = rectClip.x/10 + (0 == rectClip.x%10 ? 0 : 1);
    int jj = rectClip.y/10 + (0 == rectClip.y%10 ? 0 : 1);
    int maxi = rectClip.width/10 +
            (0 == rectClip.width%10 ? 0 : 1);
    int maxj = rectClip.height/10 +
            (0 == rectClip.height%10 ? 0 : 1);
    for( ; ii<=maxi ; ii++ )
        for( ; jj<=maxj ; jj++ )
            // draw the ii-th, jj-th image, using
            // this Component as the ImageObserver.
            g.drawImage(_aaimg[ii][jj], ii*10, jj*10,
                    this);
}
```

## getColor

**ClassName**

Graphics

**Purpose**

Gets the current rendering color of the Graphics object.

**Syntax**

public abstract Color getColor();

**Parameters**

None.

**Description**

Accesses the current foreground color of the graphics object, which is one of the internal *state* variables of the Graphics object. All graphical primitive and text rendering operations are done using the foreground color.

**Imports**

*java.awt.Color*

**Returns**

A Color object containing the relative RGB (Red/Green/Blue) values of the current foreground color of the Graphics object.

**See Also**

The setColor method of the Graphics class; the Color class

**Example**

This example uses both getColor and setColor to modify the Graphics' current rendering color.

```
public void paint(Graphics g) {

    // Get current drawing color.
    Color c = g.getColor();

    // Modify current color by switching red and blue
    // color values.
    g.setColor(new Color(c.getBlue(), c.getGreen,
            c.getRed());

    // Do drawing with the new color.
    ...
}
```

## getFont

**ClassName**

Graphics

**Purpose**

Gets the current font of the Graphics object.

**Syntax**

public abstract Font getFont();

**Parameters**

None.

**Description**

Accesses the current font of the Graphics object The current font is one of the internal state variables defining the current state of a Graphics object. All text rendering operations are done using the currentFont.

**Imports**

*java.awt.Font*

**Returns**

A Font object describing the current font in use by the Graphics object.

**See Also**

The setFont method of the Graphics class; the Font class

**Example**

This example uses getFont and setFont to make the current font for rendering text boldface.

```
public void paint(Graphics g) {
    // Get the current Font.
    Fonr f = g.getFont();

    // change the Font by making a boldface version of it.
    g.setFont(new Font(f.getName(),
```

```
        f.getStyle() | Font.BOLD, f.getSize());

    // Do something with the new font...
    ...
}
```

## getFontMetrics

**ClassName**

Graphics

**Purpose**

Returns a FontMetrics object for the Font and the display surface associated with this Graphics object.

**Syntax**

public abstract FontMetrics getFontMetrics();
public abstract FontMetrics getFontMetrics(Font f);

**Parameters**

***Font f***

A specific Font to get a FontMetrics for. The resultant FontMetrics represents the metrics of text rendered on the display surface associated with this Graphics object using Font *f*.

**Description**

The same font can actually render differently on different display surfaces, especially if those display surfaces represent very dissimilar graphical devices. A FontMetrics object describes how the font will render on a particular Graphics object's display surface. The first overloaded version of this method will generate a FontMetrics describing how the Graphics object's current font, one of the variables of the Graphic's internal state, will display on the drawing surface. The other overloaded version allows you to pass in a Font object.

**Returns**

A FontMetrics object describing how the specified font, defined by a Font object, will be displayed on the Graphic's drawing surface.

**See Also**

The getFont and setFont methods of the Graphics class

**Example**

```
public void paint(Graphics g) {
    // Get the FontMetrics for the current Font on
    // g's display surface.
    FontMetrics fm = g.getFontMetrics();

    // Display width of the string "Ginger" to System.out.
    System.out.println("Width of 'Ginger' is " +
        fm.stringWidth("Ginger"));
}
```

## scale

**ClassName**

Graphics

**Purpose**

Changes the scale of the X and Y axes of this Graphics object's coordinate system.

**Syntax**

public abstract void scale(float xScale, float yScale);

**Parameters**

***float xScale***

The new ratio of physical device units of the display surface to logical units of the Graphics object in the horizontal direction.

***float yScale***

The new ratio of physical device units of the display surface to logical units of the Graphics object in the vertical direction.

**Description**

This method allows you to modify the ratio of physical device units to logical device units in both the horizontal and vertical directions. The scale of the Graphics object is one of the internal state variables that can affect the appearance of rendered geometric primitives, text, and images on the display surface.

The physical device units of a display surface are an atomic measure of the smallest addressable surface element. For example, the physical device units of the on-screen desktop are pixels. Pixels are also the physical device units of Image objects in memory.

Changing the scale of a Graphics object attached to the on-screen desktop to, say, two would mean that every pixel on the display surface was represented by two logical units of the Graphics object. In that case, a reference to the point (10,10) in a Graphics method would actually map to a physical point 5 pixels to the right and 5 pixels below the origin on the screen.

Different graphical devices have different physical device units. The physical device units of a laser printer probably would be much smaller than those of a dot-matrix printer.

**Imports**

None.

**Returns**

None.

## setColor

**ClassName**

Graphics

**Purpose**

Modifies the current rendering color of this Graphics object.

**Syntax**

public abstract void setColor(Color c);

**Parameters**

***Color c***

A Color object containing the RGB (Red/Green/Blue) values of the color to use for the new foreground color of the Graphics object.

**Description**

Changes the current foreground color used by the Graphics object when rendering geometric primitives or text on the display surface. The current foreground color is one of the internal state variables that defines the current state of a Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The getColor method of the Graphics class; the Color class

**Example**

See the example for the getColor method of the Graphics class.

## setFont

**ClassName**

Graphics

**Purpose**

Modifies the font used for rendering text by this Graphics object.

**Syntax**

public abstract void setFont(Font f);

**Parameters**

***Font f***

A Font object describing the font to use when rendering text on the display surface using any of the Graphics class' text rendering methods.

**Description**

Changes the current font used by the Graphics object when rendering text on the display surface. The current font is one of the internal state variables that defines the current state of a Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The getFont method of the Graphics class; the Font class

**Example**

See the example for the getFont method of the Graphics class.

## setPaintMode

**ClassName**

Graphics

**Purpose**

Sets the painting mode of this Graphics object to "overwrite", as opposed to XOR mode.

**Syntax**

public abstract void setPaintMode();

**Parameters**

None.

**Description**

Changes the current painting mode used by the Graphics object when rendering geometric primitives or text on the display surface to "overwrite". When using this mode, all rendering overwrites the current display surface contents using the current foreground color. The current painting mode is one of the internal state variables that defines the internal state of a Graphics object.

**Imports**

None.

**Returns**

None.

**See Also**

The setXORmode method for the Graphics class

**Example**

This example uses both the overwrite and XOR painting modes in the same custom paint method implementation.

```
public void paint(Graphics g) {
    // Make sure painting mode is "overwrite".
    g.setPaintingMode();

    // draw a couple of boxes.
    g.fillRect(0, 0, size().width, size().height);
    g.drawRect(0, 0, size().width-1, size().height-1);

    // put Graphics into XOR mode, using white as the
    // alternate color.
    g.setXORMode( Color.white );

    // draw some more boxes. Overlapping areas will
    // be shown as white.
    g.fillrect(10, 10, size().width-20, size().height-20);
    g.drawRoundRect(10, 10, size().width-20,
        size().height-20, 10, 10);
}
```

## setXORMode

**ClassName**

Graphics

**Purpose**

Changes the Graphics object’s painting mode to XOR mode, as opposed to overwrite mode.

**Syntax**

public abstract void setXORMode(Color c);

**Parameters**

***Color c***

A Color object containing the RGB (Red/Green/Blue) values of the color to use for the alternate to the foreground.

**Description**

Changes the current painting mode used by the Graphics object when rendering geometric primitives or text on the display surface to "XOR" mode. In XOR mode, the color value of a pixel, after a rendering operation, can be determined by this formula:

```
outColor(x, y) = inColor(x, y) ⊗ drawingColor ⊗ alternateColor
```

where the ⊗ symbol denotes the bitwise XOR operation.

**Imports**

None.

**Returns**

None.

**See Also**

The setPaintingMode method of the Graphics class

**Example**

See the Ovals Applet example earlier in this chapter.

## Applet

**Purpose**

An embeddable interactive Component, suitable for embedding in World Wide Web pages using special HTML tags.

**Syntax**

public class Applet extends Panel

**Description**

A Java Applet is an interactive Component specially designed for use across the World Wide Web. The Applet class defines methods for controlling the lifetime of an Applet object, for which your applets provide custom implementations. Each applet running in an applet-aware browser has its own Thread, which uses the Applet methods init, start, stop and destroy to control the applet's lifetime. The Applet communicates with the browser through AppletContext and AppletStub objects. Figure 1-8 illustrates the inheritance relationship of the Applet class.

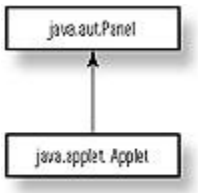


**Figure 1-8** Class diagram of the Applet class

**PackageName**
*java.applet*
**Imports**
*java.awt.**
**Constructors**
None.
**Parameters**
None.

## isActive

**ClassName**
Applet
**Purpose**
Indicates whether or not the Applet has been started.
**Syntax**
public boolean isActive();
**Parameters**
None.
**Description**
Just before the Applet's start method is called, the Applet is marked as "active". At that point, all calls to this method return true. Before that time and just before destroy is called, the Applet is marked as not active.
**Imports**
None.
**Returns**
True is returned if this method is called at any time from just before the Applet's start method is called to just before the Applet's destroy method is called.
**Example**
Check to see if Applet "Professor" is active.

```
...
Applet appletProf =
        getAppletContext().getApplet("Professor");
if(null != appletProf)
    if(appletProf.isActive())
        System.out.println("Professor is active!");
...
```

## getDocumentBase

**ClassName**
Applet
**Purpose**
Gets the URL for the document this Applet is embedded in.
**Syntax**
public URL getDocumentBase();
**Parameters**

None.

**Description**

The URL for the document this Applet is embedded in is returned. This method is a shallow wrapper around AppletStub.getDocumentBase, so if the AppletStub is not implemented then, a call to this method will cause a NullPointerException to be thrown.

**Imports**

*java.net.URL*

**Returns**

The URL pointing to the document this Applet is embedded in.

**See Also**

The getCodeBase method of the Applet class.

**Example**

```
...
System.out.println("Doc base is: " + getDocumentBase());
...
```

## getCodeBase

**ClassName**

Applet

**Purpose**

Gets the URL for this Applet's .CLASS file.

**Syntax**

public URL getCodeBase();

**Parameters**

None.

**Description**

The URL for the this Applet's .CLASS file is returned. This method is a shallow wrapper around AppletStub.getCodeBase, so if the AppletStub is not implemented, then a call to this method will cause a NullPointerException to be thrown.

**Imports**

*java.awt.URL*

**Returns**

The URL pointing to this Applet's .CLASS file.

**See Also**

The getDocumentBase method of the Applet class

**Example**

```
...
System.out.println("Code base is: " + getCodeBase());
...
```

## getParameter

**ClassName**

Applet

**Purpose**

Gets the string value of a particular Applet parameter.

**Syntax**

public String getParameter(String name);

**Parameters**

***String name***

Name of the parameter to retrieve. This is the value of the "name" tag within the HTML <PARAM> field which defines the Applet.

**Description**

This method returns one of the parameters to this Applet. Parameters are declared between the <APPLET>..</APPLET> delimiters in HTML files. The <PARAM> tag has two possible fields: "name" and "value". By indicating one of the valid names for this Applet, the corresponding "value" field string will be returned.

**Imports**

None.

**Returns**

The String associated with the parameter whose "name" field value is the *name* parameter. If no such parameter exists, then null is returned.

**See Also**

The getParameters method of the Applet class

**Example**

```
...
// Retrieve each of the Applet's parameters and print
// all their values.
String[ ][ ] aastrParams = getParameters();
for( int ii=0 ; ii<aastrParams.length ; ii++ )
    System.out.println(aastrParams[ii],
        getParameter(aastrParams[ii]));
...
```

## getAppletContext

**ClassName**

Applet

**Purpose**

Retrieves the AppletConext for this Applet.

**Syntax**

public AppletContext getAppletContext();

**Parameters**

None.

**Description**

The AppletContext represents the browser this Applet is being displayed on. To retrieve a reference to an Applet's AppletContext, use this method.

**Imports**

*java.applet.AppletContext*

**Returns**

A reference to this Applet's AppletContext is returned. Note that if the Applet is not instantiated within a proper browser, then this method will return null. That is,

if you have an application which simply creates an Applet instance, then that Applet's AppletContext will be null.

**See Also**

The getAppletStub method of the Applet class

**Example**

This example uses the AppletContext to get an array of all the applets running within the browser.

```
...
Enumeration e = getAppletContext().getApplets();
// do something with each Applet in the Enumeration...
...
```

## showStatus

**ClassName**

Applet

**Purpose**

Displays a message on the browser's status bar.

**Syntax**

public void showStatus(String msg);

**Parameters**

***String msg***

Message to be displayed on the browser's status bar.

**Description**

Browsers generally have a status bar below the main display window. Use this method to place a message within that status bar. This method is a shallow wrapper around AppletContext.showStatus. If the Applet is not created within the context of a browser which implements AppletContext, then a call to this method will throw a NullPointerException.

**Imports**

None.

**Returns**

None.

**See Also**

The showStatus method of the AppletContext class

**Example**

```
...
public void start() {
    // Show message indicating the Applet was started...
    showStatus("Applet started!");
...
}
...
```

## play

**ClassName**

Applet

**Purpose**

Downloads and plays an AudioClip from an audio data file.

**Syntax**

public void play(URL url); public void play(URL url, String str);

**Parameters**

***URL url***

URL or base of a relative URL to the audio data file for the AudioClip you want to play.

***String str***

Relative URL to the URL you want to play.

**Description**

This method is a simple shorthand for getting an AudioClip and playing it. Use of this method saves about three lines of explicit coding.

**Imports**

*java.applet.AudioClip, java.net.URL*

**Returns**

None.

**See Also**

The getAudioClip method of the Applet class; the AppletContext class; the AudioClip interface

**Example**

This example reproduces the code of the ThemeMusicApplet, provided earlier in this chapter.

```
import java.Applet.*;

public class ThemeMusicApplet extends Applet {
  AudioClip audclipTheme;

  public void init() {
    // load the audio clip.
    audclipTheme = getAudioClip( getDocumentBase(),
        "images/theme.au" );

    // shrink display surface...never used.
    resize( 0, 0 );
  }

  public void start() {
    // start the audio loop.
    audclipTheme.loop();
  }

  public void stop() {
    // halt the audio loop.
    audclipTheme.stop();
  }

  public void destroy() {
    // release the audio clip from memory.
```

```
    audclipTheme = null;
  }
}
```

## init()

**ClassName**
Applet
**Purpose**
Called by the Applet's Thread to allow it to initialize itself.
**Syntax**
public void init();
**Parameters**
None.
**Description**
The init() method is one of the four methods which define an Applet's action during its lifetime. In your custom applet, implement this method to allocate any resources you will need for your applet to run. The init() method is called only once, and always before this first invocation of the applet's start() method.
**Imports**
None.
**Returns**
None.
**See Also**
The start(), stop(), and destroy() methods of the Applet class
**Example**
See the project for this chapter, which makes extensive use of the Applet's init(), start(), stop(), and destroy() methods.

## start()

**ClassName**
Applet
**Purpose**
Called by the Applet's Thread to start it running.
**Syntax**
public void start();
**Parameters**
None.
**Description**
The start() method is one of the four methods which define an Applet's action during its lifetime. In your custom applet, implement this method to actually perform the applet's behavior. The start() method is potentially called several times during the lifetime of the applet. Each call to start() is matched by exactly one subsequent call to stop(), sometime in the future. A typical operation performed in the start() method is kick-starting the applet's background Threads.
**Imports**

None.

**Returns**

None.

**See Also**

The init(), stop(), and destroy() methods of the Applet class

**Example**

See the project for this chapter, which makes extensive use of the Applet's init(), start(), stop(), and destroy() methods.

## stop()

**ClassName**

Applet

**Purpose**

Called by the Applet's Thread to stop it running.

**Syntax**

public void stop();

**Parameters**

None.

**Description**

The stop() method is one of the four methods which define an Applet's action during its lifetime. In your custom applet, implement this method to gracefully shut down the applet. The stop() method is potentially called several times during the lifetime of the applet. Each call to stop() is matched by exactly one prior call to start(). Stop any background Threads from processing before returning from your custom implementation of this method.

**Imports**

None.

**Returns**

None.

**See Also**

The init(), start(), and destroy() methods of the Applet class

**Example**

See the project for this chapter, which makes extensive use of the Applet's init(), start(), stop(), and destroy() methods.

## destroy()

**ClassName**

Applet

**Purpose**

Called by the Applet's Thread to allow it to perform final clean-up.

**Syntax**

public void destroy();

**Parameters**

None.

**Description**

The destroy() method is one of the four methods which define an applet's action during its lifetime. In your custom applet, implement this method to deallocate any resources allocated during the applet's lifetime. The destroy() method is called exactly once, just before the Applet object is destroyed.

**Imports**

None.

**Returns**

None.

**See Also**

The init(), start(), and stop() methods of the Applet class

**Example**

See the project for this chapter, which makes extensive use of the Applet's init(), start(), stop(), and destroy() methods.

## The Applet and Graphics Project: The Game of Life

### Project Overview

The Applet and Graphics class project demonstrates a non-trivial applet, suitable for embedding in a World Wide Web page and viewing with an applet-enabled Web browser. This project illustrates animation using double-buffered screen updating to minimize "flicker", and background Thread processing to create successive animation images. As with several of the Applet code samples provided in this chapter, the Life applet uses custom implementations of the essential Applet methods init(), start(), stop(), and destroy() to manage resources and processing during the lifetime of the Life applet object.

"The Game of Life" is a simple example of artificial life on the computer, introduced by Conway in 1970. The "game" is played on a grid. Each cell on the grid is designated as either "alive" or "not-alive" (i.e., dead), termed the cell's "state". At each turn of the game, the computer determines the state of each cell in the grid based on the state of the cell and its adjacent cells in the previous turn. These are the rules for determining a cell's state:

- An alive cell remains alive in the next turn if there are exactly two or three adjacent alive cells. This is termed the "loneliness rule".
- A dead cell becomes alive in the next turn if there are exactly three adjacent alive cells. This is termed the "reproduction rule".
- All other cells, whether alive or dead, will be dead the next turn.

It is fun and interesting to watch this program build cell colonies through turn-by-turn application of the above rules. Huge clusters of cells can die out from over-population, while small clusters of five or six cells can grow into mammoth cell structures. Figures 1-

9, 1-10, and 1-11 present screenshots from three successive turns, or "generations," of a particular run through the Game of Life using the Life applet built in this project.

**Figure 1-9** Turn or generation of the Game of Life

**Figure 1-10** Turn or generation of the Game of Life

**Figure 1-11** Turn or generation of the Game of Life

Note that in the Life.java code, an underscore character "_" is prepended to all class member variables to distinguish them from function names, local variables, etc. This is to improve the readability of the code.

## Assembling the Project

**1.** Begin the file named Life.java by declaring the Life Applet with its necessary member variables and object instances:

```
import java.applet.*;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.image.ImageObserver;
import java.awt.Color;
import java.awt.MediaTracker;
import java.net.URL;

public class Life extends Applet implements Runnable,
    ImageObserver {

  /* State variables. */
  boolean[ ][ ] _a2dfGameGrid;    // Stores current generation
  int _nGeneration;               // keeps track of turn #
  boolean _fGameGridDisplayed;    // Has current grid been
displayed?
```

```
  Thread _threadNextGen;          // Calcs next generation.

  int _nxCells;                   // Game grid width.
  int _nyCells;                   // Game grid height.

  int _nxCellPixels;              // phys. width of each cell.
  int _nyCellPixels;              // phys. height of each cell.
  int _nxCellOrigin;              // phys. origin of cell 0,0.
  int _nyCellOrigin;              // phys. origin of cell 0,0.

  int _nxPixels;                  // Applet width in pixels.
  int _nyPixels;                  // Applet height in pixels.

  Image _imageAlive;               // Image to use for "alive"
                                        cells.
  Image _imageDead;                // Image to use for "not-
alive"
                                        cells.

  Image _image2ndSurface;          // Image for 2ble buffering.
  Graphics _gc2ndSurface;          // Graphics for 2ble
buffering.

  MediaTracker _medtrack;          // To track alive and dead
images.

  /* Constants */
  static final int DEF_XCELLS = 100;
  static final int DEF_YCELLS = 100;
```

**2.** The Life class' overriding implementation of imageUpdate prevents Life Applet objects from repainting during construction of Images downloaded by the Applet. Life will later explicitly handle repainting with a background Thread.

```
  /* ********
   * imageUpdate() is called when the Alive and Dead cell
   * images are being loaded. An ImageObserver-implementing
   * class is required to draw images, which this applet class
   * implements with this method.
   ******** */
  public synchronized boolean imageUpdate( Image img,
      int nFlags, int x, int y, int width, int height ) {
    return true;
  }
```

**3.** Life's init implementation reads in the applet parameters, allocates and fills in the initial generation game grid, resizes the display area of the applet, and loads the "alive" and "not-alive" images.

```
public void init() {
    /*
     ** Retrieve the Game Grid dimensions
     ** ( _nxCells x _nyCells ). If not given,
     ** use default Game Grid size.
     */
    try {
     _nxCells = Integer.parseInt(
          getParameter( "xCells" ) );
     _nyCells = Integer.parseInt(
```

```
        getParameter( "yCells" ) );
    } catch (Exception e) {
   _nxCells = DEF_XCELLS;
   _nyCells = DEF_YCELLS;
  }

  /*
   ** Retrieve the physical display size
   ** ( _nxPixels x _nyPixels ). If not given,
   ** use the Game Grid size ( _nxCells x _nyCells ).
   */
  try {
   _nxPixels = Integer.parseInt(
        getParameter( "xPixels" ) );
   _nyPixels = Integer.parseInt(
        getParameter( "yPixels" ) );
 } catch (Exception e) {
   _nxPixels = _nxCells;
   _nyPixels = _nyCells;
 }

 /*
  ** Calculate the physical cell size.
  */
 _nxCellPixels = _nxPixels / _nxCells;
 _nyCellPixels = _nyPixels / _nyCells;
 _nxCellOrigin = ( _nxPixels / 2 ) -
        ( ( _nxCellPixels * _nxCells ) / 2 );
 _nyCellOrigin = ( _nyPixels / 2 ) -
        ( ( _nyCellPixels * _nyCells ) / 2 );

 /*
  ** Create the initial Game Grid, and fill it
  ** with the initial pattern of cells.
  ** Note that setGameGrid automatically repaints
  ** the applet.
  */
 _fGameGridDisplayed = false;
 _a2dfGameGrid = placeInitialPatternToGrid(
        createNewGameGrid( _nxCells, _nyCells ),
       _nxCells, _nyCells );
 _nGeneration = 0;

 /*
  ** Create the 2nd surface for double-buffered
  ** drawing.
  */
 _image2ndSurface = createImage( _nxPixels, _nyPixels );
 _gc2ndSurface = _image2ndSurface.getGraphics();
 _gc2ndSurface.setColor( Color.black );
 _gc2ndSurface.setPaintMode();

 /*
  ** Create the media tracker, and start it loading
  ** the alive and dead images.
  */
  _medtrack = new MediaTracker( this );
```

```
    _medtrack.addImage( getAliveImage(), 0 );
    if( null != getDeadImage() )
      _medtrack.addImage( getDeadImage(), 1 );
    return;
  }
```

**4.** Life's start method creates the Calc Thread, using this object's run() method to do the successive game grid generation.

```
public synchronized void start() {
    /*
     ** Create the Calc thread and start it running.
     */
    _threadNextGen = new Thread( this );
    _threadNextGen.start();
    return;
  }
```

**5.** The Calc Thread, created and started in Life's start() method implementation, is made to halt in stop(). The run() method (below) is written so that the Calc Thread will stop running when the _threadNextGen member variable is null. Note that both the start() and stop() methods are synchronized to prevent simultaneous execution by more than one Thread.

```
public synchronized void stop() {
    /*
     ** Remove references to the Calc thread, which will
     ** cause it to stop processing very soon. The
     ** resume wakes up the (potentially) suspended
     ** Calc thread.
     */
    Thread threadTempRef = _threadNextGen;
    _threadNextGen = null;
    threadTempRef.resume();

    return;
  }
```

**6.** Life's Calc Thread, which calculates successive generations of the game grid, executes the run() method. The Calc Thread drops out of the continuous loop when _threadNextGen is set to null in Life.stop(). Once a new game grid is created, Life's setGameGrid() method is called to update the display.

```
public void run() {
    boolean[ ][ ] a2dfNewGrid = null;
    int[ ][ ] a2dnSums = new int[ _nxCells ][ _nyCells ];
    int nI;
    int nJ;

    /*
     ** Make sure the alive and dead images have
     ** been loaded before really doing anything
     ** in this thread.
     */
    try {
       _medtrack.waitForID( 0 ); // the alive image
       if( null != getDeadImage() )
         _medtrack.waitForID( 1 ); // the dead image.
  } catch (Exception e) {}
```

```
/*
 ** This continuous loop generates iterative
 ** generations of Life.
 */
while( null != _threadNextGen ) {
  /*
   ** Allocate a new game grid only if current
   ** game grid is using the last one we
   ** allocated, or if we never allocated one
   ** before here.
   */
  if( ( null == a2dfNewGrid ) ||
       ( _a2dfGameGrid == a2dfNewGrid ) )
     a2dfNewGrid = createNewGameGrid( _nxCells,
         _nyCells );

  /*
   ** Clear out the Sums grid for this iteration.
   */
  for( nI=0 ; nI<_nxCells ; nI++ )
    for( nJ=0 ; nJ<_nyCells ; nJ++ )
      a2dnSums[ nI ][ nJ ] = 0;

  /*
   ** To calc next generation: run through current
   ** generation: for each "alive" cell, do this:
   **  Add 1 to the Sums grid of each adjacent cell.
   **  Add 9 to the Sums grid of this cell.
   ** When we're done, only cells with Sum of
   ** 3 (dead w/ 3 adjacent), 11 (alive w/
   ** 2 adjacent), or 12 (alive w/ 3 adjacent)
   ** will be alive in the new grid.
   */
  for( nI=0 ; nI<_nxCells ; nI++ )
     for( nJ=0 ; nJ<_nyCells ; nJ++ )
       if( _a2dfGameGrid[ nI ][ nJ ] ) {
         for( int nII=-1 ; nII<2 ; nII++ )
           if( ( 0 <= nI + nII ) &&
               ( _nxCells > nI + nII ) )
             for( int nJJ=-1 ; nJJ<2 ; nJJ++ )
               if( ( 0 <= nJ + nJJ ) &&
                   ( _nyCells > nJ + nJJ ) )
                 a2dnSums[ nI+nII ]
                     [ nJ+nJJ ] += 1;
           a2dnSums[ nI ][ nJ ] += 8;
       }

       for( nI=0 ; nI<_nxCells ; nI++ )
         for( nJ=0 ; nJ<_nyCells ; nJ++ )
           switch( a2dnSums[ nI ][ nJ ] ) {
             case 3:
             case 11:
             case 12:
               a2dfNewGrid[ nI ][ nJ ] = true;
               break;
             default:
               a2dfNewGrid[ nI ][ nJ ] = false;
```

```
            }
          /*
           ** Make sure the current game grid
           ** has been displayed before updating
           ** the game grid with the next generation.
           */
          while( ( ! setGameGrid( a2dfNewGrid ) ) &&
              ( null != _threadNextGen ) )
            try {
              _threadNextGen.suspend(); // this thread
            } catch (Exception e) {}
      }

      return;
  }
```

**7.** Paint() is called by the Java system asynchronously whenever the Life Applet must be updated on screen. This can be in response to an explicit call to Life.repaint() (done in setGameGrid()), or in response to a windowing event such as the Life applet being scrolled off and then back on screen. This implementation of paint uses the current game grid to place multiple copies of the "alive" and "not-alive" images to an in-memory Image. When the in-memory Image has been updated, it is copied to the screen. The update() method is overriden to call paint() without doing anything else.

```
public synchronized void paint(Graphics g) {
    int nI;
    int nJ;

    /*
     ** Display the grid on the applet surface. This
     ** involves just running through all the cells
     ** and placing the corrosponding image on the
     ** display surface,
     */
    _gc2ndSurface.fillRect( 0, 0, _nxPixels, _nyPixels );

    for( nI=0 ; nI<_nxCells ; nI++ )
      for( nJ=0 ; nJ<_nyCells ; nJ++ )
        if( _a2dfGameGrid[ nI ][ nJ ] )
          _gc2ndSurface.drawImage( getAliveImage(),
              ( nI * _nxCellPixels ) + _nxCellOrigin,
                  ( nJ * _nyCellPixels ) + _nyCellOrigin,
               this );
        else if ( null != getDeadImage() )
          _gc2ndSurface.drawImage( getDeadImage(),
              ( nI * _nxCellPixels ) + _nxCellOrigin,
                  ( nJ * _nyCellPixels ) + _nyCellOrigin,
              this );
    g.drawImage( _image2ndSurface, 0, 0, this );

    _fGameGridDisplayed = true;
    /*
     ** Use resume() to release the (potentially)
     ** suspended Calc thread.
     */
    _threadNextGen.resume();
```

```
    return;
  }

public void update(Graphics g) {
    paint(g);
  }
```

**8.** setGameGrid() is called by the Calc Thread after it has completed calculating the next generation. This is synchronized so paint(), start(), or stop() cannot be entered while the current game grid is being updated.

```
private synchronized boolean setGameGrid(
    if( ! _fGameGridDisplayed )
        return false;

    try {
      System.arraycopy( a2dfGrid, 0, _a2dfGameGrid,
          0, _nxCells );
    } catch(Exception e) {
    _a2dfGameGrid = a2dfGrid;
    }

    _fGameGridDisplayed = false;
    _nGeneration++;
    repaint();
    return true;
  }
```

**9.** Life implements several utility methods to make the code in the previous steps more readable.

```
  /* ********
   * createNewGameGrid( x, y ) allocates and returns a reference
   * for a 2d array of booleans.
   ******** */
  private boolean[ ][ ] createNewGameGrid( int xCells, int yCells
) {
    boolean[ ][ ] a2dfNewGrid = new boolean[ xCells ][ yCells ];
    return a2dfNewGrid;
  }

  /* ********
   * placeInitialPatternToGrid( boolean[ ][ ] ) will read in
   * the initial cell pattern from the three applet
   * parameters "xStartPatternCells", "yStartPatternCells",
   * and "strStartPattern". The start pattern will be
   * placed centered on the 2d array passed in.
   ******** */
  private boolean[ ][ ] placeInitialPatternToGrid(
      boolean[ ][ ] a2dfGrid, int nxGridCells,
      int nyGridCells )
  {
    /*
     ** Get the dimensions of the starting pattern.
     ** xStartPatternCells and yStartPatternCells
     ** are not optional parameters.
     */
    int nxStartCells = Integer.parseInt(
        getParameter( "xStartPatternCells" ) );
```

```
  int nyStartCells = Integer.parseInt(
      getParameter( "yStartPatternCells" ) );

  /*
   ** Calculate the X and Y cell offsets to
   ** begin placing the initial pattern on
   ** the grid.
   */
  int nxPatternOffsetCells = ( nxGridCells / 2 ) -
      ( nxStartCells / 2 );
  int nyPatternOffsetCells = ( nyGridCells / 2 ) -
      ( nyStartCells / 2 );

  /*
   ** Retrieve the start pattern descriptive string.
   */
  String strStartPattern =
      getParameter( "strStartPattern" );

  /*
   ** For each cell in the starting pattern, update
   ** the corresponding grid cell.
   */
  int iPatternString = 0;
  for( int nI=0 ; nI<nxStartCells ; nI++ )
    for( int nJ=0 ; nJ<nyStartCells ; nJ++ ) {
      int iNext0 = strStartPattern.indexOf(
          '0', iPatternString );
      int iNext1 = strStartPattern.indexOf(
          '1', iPatternString );

      if( ( iNext0 < iNext1 ) &&
          ( iNext0 != -1 ) )
        iPatternString = iNext0 + 1;
      else if( iNext1 != -1 )
        iPatternString = iNext1 + 1;
      else {
        nI = nxStartCells;
        nJ = nyStartCells;
        continue;
    }

    a2dfGrid[ nI+nxPatternOffsetCells ]
        [ nJ+nyPatternOffsetCells ] =
            ( strStartPattern.charAt(
                iPatternString - 1 ) ==
                    '1' );
  }

  return a2dfGrid;
}

/* ********
 * getAliveImage()
 * getDeadImage()
 * These methods are responsible for identifying and loading
 * the "alive" and "not-alive" cell images. The alive image
```

```
   * is the only one required. The "urlAliveImage" parameter
   * holds a relative URL to the "alive" cell image, and the
   * "urlDeadImage" parameter holds a relative URL to the
   * "not-alive" cell image.
   ******** */
  private Image getAliveImage() {
    if( null == _imageAlive ) {
        URL urlAliveImage = null;
      try {
        urlAliveImage = new URL( getDocumentBase(),
            getParameter( "urlAliveImage" ) );
      } catch (Exception e) {}
      _imageAlive = getImage( urlAliveImage );
      }

  return _imageAlive;
}

  private Image getDeadImage() {
    if( null == _imageDead ) {
      if( null == getParameter( "urlDeadImage" ) )
        return null;

      URL urlDeadImage = null;
      try {
        urlDeadImage = new URL( getDocumentBase(),
            getParameter( "urlAliveDead" ) );
      } catch (Exception e) {}
      _imageDead = getImage( urlDeadImage );
    }

    return _imageDead;
  }
};
```

**10.** Enter the following HTML code into a file named EXAMPLE1.HTML in the same directory as Life.java:

```
<HTML>
<HEAD>
<TITLE>Life Applet Example</TITLE>
</HEAD>
<BODY>
<H1>Life Applet Example</H1>
Below is the Game of Life applet.
<HR>
<CENTER>
<APPLET CODE="Life.class" WIDTH=300 HEIGHT=300>
<PARAM NAME="xPixels" VALUE="300">
<PARAM NAME="yPixels" VALUE="300">
<PARAM NAME="xCells" VALUE="30">
<PARAM NAME="yCells" VALUE="30">
<PARAM NAME="urlAliveImage" VALUE="alive.gif">
<PARAM NAME="xStartPatternCells" VALUE="10">
<PARAM NAME="yStartPatternCells" VALUE="10">
<PARAM NAME="strStartPattern"
        VALUE="1111111100
               1100000011
```

```
                1100000011
                1111111100
                1111111100
                1100000011
                1100000011
                1100000011
                1111111100
                1111111100">
</APPLET>
</CENTER>
<HR>
Here's <A HREF="Life.java">the source</A>
</BODY>
</HTML>
```

## How It Works

Table 1-5 lists the applet parameters, both required and optional, used by the Life applet.

**Table 1-5** Life applet parameter descriptions

| Parameter | Required | Description |
|---|---|---|
| xCells, yCells | Yes | The number of columns and rows, respectively, of the grid of cells to be displayed by the Life applet. |
| xPixels, yPixels | Yes | The physical size of the applet's display surface in pixels. |
| urlAliveImage | Yes | A URL pointing to an image the applet is to use to represent alive cells. |
| urlDeadImage | No | A URL pointing to an image the applet is to use to represent dead cells. If this parameter is not provided, the applet displays nothing in dead cells. |
| xStartPattern Cells and yStartPatternCells | Yes | The number of columns and rows, respectively, of the initial pattern of cells described by the strStartPattern parameter. |
| strStartPattern | Yes | A string of "1" and "0" characters describing the initial states of the grid of cells. This string should have exactly (xStartPatternCells * yStartPatternCells) "1" or "0" characters in it. All other characters in the string are ignored. The string is interpreted as a left-to-right, top-to-bottom list of cell states. The initial pattern is |

centered on the Life grid automatically.

Three important techniques are used by the Game of Life applet:

- Double-buffering to ensure smooth visual transition between successively displayed frames.
- Overriding Component.update() to avoid "flicker".
- Background processing to generate successive animation frames.

## Double-Buffered Rendering

Life's paint method has the responsibility of displaying a grid of MxN cells. One way this could be accomplished is by simply rendering each cell on the applet's drawing surface is a nested *for* loop:

```
public void paint(Graphics g) {
    for( nI=0 ; nI<M ; nI++ ) {
        for( nJ=0 ; nJ<N ; nJ++ ) {
            displayCell( nI, nJ, g );
        }
    }
}
```

The big problem with this method of display is that, especially for large M and N, the user will see each individual row of the display surface get updated. For less jerky animation, the successive frames should simply "pop" onto the screen, fully rendered. That's what "double-buffered rendering" does: It allows you to update the display surface all at once, instead of little-by-little.

In double-buffered rendering, an Image object is created in memory with the exact same dimensions as the applet's display surface. All rendering is done to a Graphics object attached to that in-memory Image object. When all rendering is completed, the Image object is copied to the display surface all at once. This has the practical effect of having the on-screen display updated instantaneously, instead of little-by-little.

In the Life applet, an in-memory Image, _image2ndSurface, is created during init with the same dimensions as the applet's display surface. A Graphics object, _gc2ndSurface, is created attached to the in-memory Image, like this:

```
public void init() {
    // ...
    _image2ndSurface = createImage( _nxPixels, _nyPixels );
    _gc2ndSurface = _image2ndSurface.createGraphics();
    // ...
}
```

In paint(), the individual Life generations (each generation is an animation frame) are rendered to the _gc2ndSurface Graphics object. When the rendering is complete, the entire _image2ndSurface is copied to the applet's display surface.

## Overriding Update()

As hinted earlier in this chapter, the Java runtime system will automatically erase an applet's drawing surface before paint() is called. For nonanimation sequences, this might not be a bad thing to do, but for fast screen updating it can prove to be quite annoying to look at. Between each two frames appears a brief "flicker" when the background is erased. The code for the default implementation of Component.update, which is responsible for the "flicker" problem, looks like this:

```
public void update(Graphics g) {
    // ...
    g.fillRect(0, 0, width, height);
    // ...
    paint(g);
}
```

To reduce this flicker problem, the Life applet implements its own update method to override the default implementation it inherits from Component. The overriding implementation does not call fillRect, so the background is not erased. The custom implementation looks like this:

```
public void update(Graphics g) {
    paint(g);
}
```

## Animation Techniques

Two different animation techniques are the opposite poles of a continuum of implementations for animation in Java:

- Timesliced animation
- Computed frame animation

The simplest animation technique using Java timeslice animation, involves creating a background Thread to "timeslice", or sleep for some quanta of time before waking up and repainting the drawing surface. This is the "simplest" method because it involves the least amount of coding. To implement this animation technique, you need:

- An ordered Vector or array of Image objects, each one a frame to display.
- A "current Image object" variable, which keeps track of which frame is currently being displayed.
- A background Thread object which wakes up periodically, advances the "current Image object" indicator to the next frame, and forces the applet to repaint itself.

The JDK includes a generic Animator applet which uses the timeslicing technique to perform animation. Through its parameters, you can customize the Animator applet to display any number of frames, in any order, and even sequence sound with each frame.

The drawback of this animation technique is that it requires all frames of the animation sequence to have already been rendered onto Image objects in memory. For applets or applications which must compute and render each frame separately, such as the Life applet, the timeslicing technique is inadequate. Life uses the "computed frame" technique of animation.

The computed frame technique works by using a background Thread to compute sufficient information to render each frame "on the fly". In a continuous loop, the animation Thread computes a frame, and tells the drawing surface to display it, computes a frame, displays it, etc. In this technique, the time between the display of each frame is not necessarily constant, as in timesliced animation. Instead, the time between successive frames is dependent on how long it takes to compute and render each frame.

It is in Life's run method that each successive generation of the Life game is computed. The actual computation involves keeping an accumulated sum for each cell in the target generation grid. For each generation, run adds values to this accumulator using these two rules:

- One is added to the accumulator of each cell adjacent to an "alive" cell.
- Nine is added to the accumulator of each "alive" cell.

Based on the rules of the Game of Life presented above, only cells with an accumulated value of 2, 3, or 12 will be alive in the next generation.

The most important aspect of the run method, however, is how it is sychronized with the rendering. The synchronization is necessary to prevent a newly computed Life generation, stored in a 2D grid of boolean values, from overwriting the grid currently being rendered by the paint method.

It is conceivable under the computed frame technique for there to be a backlog of unrendered frames. This will occur if the time it takes the background Thread to compute new frames is less than the time it takes to actually render frames. In such a case, the background Thread will generate more frames than can be rendered in the same amount of time. Without proper synchronization, this could lead to frames being skipped, or other problems.

The Life applet ensures these problems won't occur by synchronizing access to the current generation grid. The background Thread will automatically suspend itself if it attempts to overwrite the current generation grid before it has been rendered on the drawing surface. A more sophisticated animation applet would utilize a synchronized storage device for storing any backlog of unrendered grids.

# Chapter 2
# The Component Class

All visual elements of a graphical interface have functionalities in common. Top-level windows, visual controls such as text boxes and push buttons, as well as simple elements for drawing images on the screen have a commonality of capabilities. The Component class, which implements these common functionalities, is an ancestor class for all graphical interface elements.

In the Java system, all classes that implement graphical interface elements are subclasses of the Component class. There are several families of Component class methods, which allow you to control the internal state and on-screen appearance of all Components. They cover the following areas of functionality:

- Component hierarchy. Components are placed on the screen within special Container components. Containers may be placed within other Containers, and so on, forming an on-screen hierarchy of Components.
- Component positioning and sizing.
- Common Component states. All Components share a basic set of internal state variables. The Component class implementation provides methods that allow these state variables to be polled and modified.
- On-screen rendering.
- Delivering and handling events. These include user, custom, and system events such as mouse events, keyboard and keyboard focus events, and so on.
- Preparing and displaying images.

In addition to the Component classes included in the *java.awt* package, you can create your own custom Components. You can create almost any imaginable visual element as a custom Component. The Project for this chapter demonstrates the creation of a relatively simple custom Component called a Hotspot.

## Component Hierarchy

Each Component object instance is "owned by" a parent Component object. The on-screen positioning of a Component is restricted to being within the bounds of its parent Component. More specifically, the rectangle of actual display device pixels, or "bounding rectangle," dedicated to a particular Component is restricted to lying completely within the bounding rectangle of its parent Component. Figure 2-1 illustrates the hierarchy of Components and Container components of a simple graphical user interface. The graphical Component controls are contained within Containers, which are in turn contained within the Frame window, another type of Container component. The Frame window is a top-level window, and so does not have a parent Container.

**Figure 2-1** Component hierarchy of a simple dialog box

Components that can contain other Components are derived from the Container class. There are several areas of interest specific to the Container class. This chapter will cover a minority of those topics as necessary to understand the Component class concept.

The getParent method provides a reference to the Container of any Component object:

```
Container parent = myComponent.getParent();
```

Components that have not been placed within a parent Container, obviously, will not have a parent Container object reference returned by getParent. For orphan Components, as well as for top-level Frame windows, a null will be returned by this method. Components are placed within Containers using the add method of class Container. The specifics of the overloaded versions of this method are discussed in Chapter 3. For simplicity's sake, you can assume that a call to this method effectively sets the owner of the Component object to the specific Container. Listing 2-1 adds a single push button (a type of Component object) to the interface of an Applet (a type of Container object).

**Listing 2-1** Adding a Component to a Container using the Container's add method

```
public class MyApplet extands Applet {
    ...

    public void init() {
        ...

        add("OK", new Button("OK") );

        ...
    }

    ...
}
```

## Component Positioning

All Component objects have a rectangle of display area in which they render themselves. This rectangle is called the Component's *bounding rectangle*. The size of the Component's bounding rectangle can be looked up using the Component's size method. The resize method allows you to modify these dimensions:

```
Dimension dimComp = myComponent.size();
myComponent.resize(dimComp.width + 10, dimComp.height + 10);
```

Note that the actual rectangle of screen real estate a Component is allowed to render itself on is the intersection of the component's bounding rectangle with the parent Container's

bounding rectangle (which is intersected with its parent's bounding rectangle, and so on). Therefore, if a Component has been resized to be larger in dimension than its parent Container, then the Component will be "clipped" on the screen according to its position relative to its parent Container.

For the most part, the positioning of a Component object within its parent Container is under the control of the Container's LayoutManager object. Chapter 3 discusses how Components are laid out within a Container by the LayoutManager object. A Component object is positioned relative to the upper-left corner of its parent Container. The location method returns the coordinates of a Component relative to the upper-left corner of its parent Container. That is, the upper-left corner of the parent Container is (0,0) for all Component positioning coordinates. The move method is called to change the position of a Component relative to its parent Container:

```
// Move myComponent 10 pixels right and down.
Point ptLoc = myComponent.location();
myComponent.move(ptLoc.x + 10, ptLoc.y + 10);
```

A Component's bounding rectangle can be fully described by its position and dimensions. The bounds method returns a Rectangle object whose *x* and *y* members indicate the position of the Component, and whose width and height members describe the dimensions of the bounding rectangle. The reshape method allows you to modify both the position and dimensions of a Component's bounding rectangle. Listing 2-2 demonstrates the positioning methods for Components. It centers a Component relative to its parent Container's bounding rectangle.

**Listing 2-2** Centering a Component with respect to its parent's bounding rectangle

```
Component comp;

// ... Comp is set to be a reference to a Component ...

Rectangle rectCompBounds = comp.bounds();
Dimension dimParent = comp.getParent().size();

rectCompBounds.x = (dimParent.width / 2) -
        (rectCompBounds.width / 2);
rectCompBounds.y = (dimParent.height / 2) -
        (rectCompBounds.height / 2);

comp.reshape(rectCompBounds.x, rectCompBounds.y,
        rectCompBounds.width, rectCompBounds.height);
```

## Common Component States

All basic visual elements such as push buttons, list boxes, and check boxes can either be enabled or disabled. By default, Components are enabled, though they may be disabled.

Disabled Components generally take on a “hampered mode” look and feel, and are generally unreactive to user actions like mouse clicks or keyboard input. Figure 2-2 illustrates several basic visual elements when enabled and disabled. The disable method disables a Component object, and enable forces a Component to be enabled. The isEnabled method returns a boolean true or false, indicating whether the Component is currently enabled.

**Figure 2-2** Enabled and disabled Button, List, and Choice objects

Components can also be hidden or visible. By default, Components when created are visible, but they can be hidden using hide. A hidden Component is effectively removed from the visual interface, as are all of that object’s child Components. The Component positioning and other internal state member methods act exactly the same for a Component whether the Component is hidden or visible. The show method forces a Component to be visible. The Component’s isVisible method returns a boolean true or false, indicating whether the Component is currently visible. Hiding a Component can be an effective method for removing inappropriate visual elements from the graphical interface.

The isShowing method tells you whether or not a Component has any display surface real estate assigned to it. That is, isShowing returns true only if the Component is visible, and is positioned such that its bounding rectangle intersected with its parent’s bounding rectangle is non-null.

A Component can also be marked as valid or invalid. The state of the validation flag indicates whether or not the Component must be laid out using the Component’s layout method. The default implementation of this method actually does nothing. However, the Container class overrides the layout method to actually arrange any child Components on the screen.

Use the invalidate method to mark the Component as invalid. The validate method will call layout if the state of the Component is invalid (i.e., invalidate was called prior to the call to validate). If the Component has not been marked as invalid, then validate returns without doing anything.

## On-Screen Rendering

All the basic visual Components, such as list boxes and push buttons, are able to render themselves on the screen. You can also create custom controls such as gas gauges, spin dials, or just about any visual element imaginable. The SuperBible project for this chapter illustrates the creation of just such a custom control. Your custom controls, however, must render themselves. To actually render custom controls you must re-implement one or more Component class methods.

The central method used to render a Component on the screen is paint. The paint method is passed a Graphics object attached to the display device, and clipped to the bounding rectangle of the Component. (Chapter 1, *Applets and Graphics,* discussed the Graphics class in detail and how Graphics objects are used to paint on a drawing surface.) The simplest custom Component classes re-implement this method to render the Component in the graphical interface. Listing 2-3 shows a trivial custom paint method implementation that simply draws a filled oval within the Component's bounding rectangle.

**Listing 2-3** A simple custom Component

```
class MyComponent extends Canvas {
    public MyComponent() {
        super();
    }
    public void paint(Graphics g) {
        g.fillOval( bounds() );
    }
}
```

Note that the custom Component class MyComponent actually is derived from the Canvas class. You cannot derive a class from the Component class directly, since the Component class has no public constructors. Generally, you will create custom Component classes, which are derived from the Canvas class, since the Canvas class is the simplest Component with a public constructor.

The Java system calls a Component's paint method asynchronously whenever it determines a Component object must be re-drawn on the display surface. This call is performed by a Thread created and controlled by the Java system. The Java system manages the Graphics object passed to paint directly because graphical device contexts are a limited resource in most graphical operating systems. You must not call paint directly. To force a repainting of a Component object, use the Component's repaint method. Using this method, you can schedule a repainting of the Component within a specific time period, and you can also restrict the repainting to a subset of the the Component's full bounding rectangle. There are four overloaded versions of the repaint method:

```
repaint();
repaint(lMillisecs);
repaint(x, y, width, height);
repaint(lMillisecs, x, y, width, height);
```

The two versions, which do not specify a time limit, instruct the Java runtime system to schedule a repainting of the Component *at some time in the future*. The Java system may not schedule a repainting for ten minutes, or the repainting could happen instantaneously. Repainting is a relatively low-priority operation, so the system waits until there is a lull in processing to actually perform the repainting. Using one of the two versions of the repaint method that take a *lMillisecs* parameter, you can specify a maximum number of

milliseconds the Java system can wait before forcing a repainting of your Component object.

The repaint method instructs the Java runtime system to schedule an asynchronous call to Component.update. The update method is responsible for calling paint. The default implementation of update erases the entire drawing surface of the Component object using the Component's background Color, then selects the Component's foreground Color into the Graphics object before calling paint. Figure 2-3 illustrates how Component rendering is accomplished through the three cooperating methods: paint, repaint, and update. You can see that the only method a custom Component needs to re-implement is paint. Re-implementing the update method can be quite useful, especially when animation techniques are used. A discussion of animation techniques and the update method is included in Chapter 1.

**Figure 2-3** Cooperative methods paint, repaint, and update used to keep on-screen rendering of a Component up-to-date

## Delivering and Handling Events

An *event*, in Java lingo, is an object that describes some specific occurrence in the system. For example, there are several types of mouse events to describe a user's mouse actions. There are also several types of keyboard events to describe user keyboard interactions. Event objects are created by the Java runtime system whenever a specific occurrence is detected, and these Event objects are delivered to specific Components through the Component class Event delivery and handling methods.

The Component class Event delivery methods implement a system whereby Events are passed from a Component to its parent Container, to that object's parent Container, and so on until the event is "handled."

The delivery system can best be illustrated through an example. Figure 2-4 is a screenshot of a very simple user interface in an Applet run within the JDK's AppletViewer. This interface is comprised of the Applet object itself, and a Button object with the caption OK. Imagine the user clicks on the OK Button. This causes an Event object of type ACTION_EVENT to be generated by the Java system and delivered to the OK Button. Button objects, by default implementation, do not handle this type of Event, and so the Event is further delivered to the Button's Container—the Applet object. The Applet object may or may not handle this Event. If not, the Event will further be

delivered to the Applet's Container, and so on until either the Event is handled or a top-level Container is reached.


**Figure 2-4** A very simple Applet interface

Delivering an Event to a Component is done using the Component's postEvent method. The Java runtime system delivers mouse, keyboard, or other Events to a specific Component using this method.

You can also create your own custom Events (objects derived from the Event class) and deliver them to Components using a similar mechanism. Instead of calling the postEvent method directly, call Component.deliverEvent. The default implementation of deliverEvent takes the Event and passes it to postEvent. Thus, the deafult implementation of deliverEvent is a simple wrapper for postEvent.

The postEvent method is responsible for finding an object to handle each Event it is passed. postEvent offers the Event to three different objects. If no object handles the Event, postEvent returns benignly and the Event is forgotten. The three objects postEvent offers each event to are (in order)

- The Component's peer, through peer.handleEvent
- The Component itself, through this.handleEvent
- The Component's parent Container, through parent.postEvent

The handleEvent method returns a boolean true or false value, indicating whether or not the Event was handled. This pseudo-code in Listing 2-4 illustrates the simple algorithm used by postEvent to find an object to handle each event passed to a particular Component.

**Listing 2-4** Pseudo-code for postEvent

```
boolean method postEvent(Event evt):
    1. if peer.handleEvent(evt) == true, return true.
    2. if this.handleEvent(evt) == true, return true.
    3. if parent.postEvent(evt) == true, return true.
    4. return false.
end method postEvent
```

The default implementation of handleEvent is a giant switch statement. Each Event is classified according to the type of Event, and an appropriate Component class handling method is called. For example, the Component class method mouseDown is called by the default implementation of handleEvent whenever a MOUSE_DOWN Event occurs. The summary section on the next page details all the Component class Event handling

methods. Custom Component implementations should override these methods to handle specific types of Events.

## Preparing Images for Display

Before an Image object can be rendered onto any drawing surface, a representation of the Image suitable for painting on that surface must be constructed by the Java system. The construction is an asynchronous process carried out by the Java system because this process may include downloading of Image data from a remote server. (Downloading of any kind of data is always an asynchronous operation.) The Component class includes methods to manage the Image construction process so that any Component object may prepare and display Images.

The Image construction process is started by a call to the Component's prepareImage method. The Image object to prepare is passed as a parameter to this function. To receive asynchronous notification of the progress of Image construction, an object must implement a special interface called *java.awt.image.ImageObserver*. The ImageObserver's imageUpdate method, the only method defined by the ImageObserver interface, is called by the Java system automatically as the Image object construction process proceeds.

An implementation of imageUpdate is included in the Component class, so any Component object may be used as an ImageObserver. The default implementation of this method schedules an asynchronous repainting of the Component when the Image has been prepared sufficiently to display. The code snippet in Listing 2-5 illustrates how any Component can prepare an Image for display.

**Listing 2-5** Preparing an Image for display

```
class MyComponent extends Component {
    Image _img;

    // set the img variable to a reference to an Image object
    // in the object's constructor.

    ...

    public void myPrepareMethod() {
        prepareImage(_img, this);
    }

    // This re-implementation of Component.imageUpdate() detects
    // when an Image has been fully prepared for rendering.
    public void imageUpdate(Image img, int flags, int x, int y,
        int width, int height) {
        super.imageUpdate(img, flags, x, y, width, height);
        if( flags & ImageObserver.ALLBITS )
            System.out.println( "Image is completely prepared." );
    }
}
```

## Summary of the Component Methods

Table 2-1 lists all the methods of the Component class, and provides a brief description of each. The methods are broken down by functional grouping rather than alphabetically.

**Table 2-1** Summary of Component methods

| Group | Method | Description |
|---|---|---|
| Event Handlers | action | Handles ACTION_EVENT Events. Button pushes and menu bar selections are two types of user interactions. |
| | lostFocus | Handles LOST_FOCUS Events. The keyboard focus has been removed from this Component, as when the user hits the TAB key in a dialog. |
| | gotFocus | Handles GOT_FOCUS Events. The keyboard focus has been moved to this Component, as when the user hits the TAB key in a dialog. |
| | keyDown | Handles KEY_DOWN Events. For the Component with keyboard focus, each keypress by the user creates such an Event. |
| | keyUp | Handles KEY_RELEASE Events. For the Component with the keyboard focus, each time a key is released such an Event is created. |
| | mouseDown | Handles MOUSE_DOWN Events. The user has clicked the mouse button within the Component's bounding rectangle. |
| | mouseDrag | Handles MOUSE_DRAG Events. The user has moved the mouse with the mouse button held down. |
| | mouseEnter | Handles MOUSE_ENTER Events. The mouse cursor has been moved from outside the Component's bounding rectangle to within it. |
| | mouseExit | Handles MOUSE_EXIT Events. The mouse cursor has been moved from within the Component's bounding rectangle to outside it. |
| | mouseMove | Handles MOUSE_MOVE Events. The mouse has been moved without the mouse button being held down. |
| Size and Position | bounds | Returns a Rectangle object whose *x* and *y* |

| | | |
|---|---|---|
| | | members indicate the position of the upper-left corner of the Component relative to the origin of the parent Container's origin. The Rectangle's *width* and *height* members hold the Component's dimensions. |
| | inside | Tells whether or not a particular point lies within the Component's bounding rectangle. |
| | locate | Returns a reference to the Component or subComponent that contains the indicated point. |
| | location | Returns a Point object whose *x* and *y* members indicate the position of the Component's upper-left corner relative to the parent Container's origin. |
| | move | Moves the upper-left corner of the Component to the indicated point relative to the parent Container's origin. |
| | resize | Changes the dimensions of the Component object's bounding rectangle. |
| | size | Returns a Dimension object whose *width* and *height* members indicate the size of the Component's bounding rectangle. |
| | reshape | Changes the Component's position and dimensions in a single call. The move and resize methods are actually wrappers around this method. |
| | minimumSize | Override this method to specify the smallest size a parent Container should dedicate to this Component. |
| | preferredSize | Override this method to specify the preferred size a parent container should dedicate to this Component. |
| Visual State | isValid | Indicates whether or not the Component has been marked as valid. |
| | invalidate | Marks the Component as invalid. |
| | validate | If the Component is invalid, calls layout before resetting the valid flag to valid. |
| | isVisible | Indicates whether or not the Component is currently marked as visible. |
| | show | Makes the Component visible. |
| | hide | Makes the Component invisible. |
| | isShowing | Returns true only if the Component is visible and owns some rectangle of the desktop. |

| | | |
|---|---|---|
| | isEnabled | Indicates whether or not the Component is enabled. |
| | enable | Enables the Component. |
| | disable | Disables the Component. |
| Graphics State | getForeground | Gets the Color object for the Component's foreground color. The Graphics passed to paint uses this color as its current foreground color. Uses parent's foreground if Component's color has not been set using setForeground. |
| | setForeground | Sets the current drawing color for the Graphics passed to paint. |
| | getBackground | Gets the color used to erase the Component in update. Returns parent's background color if Component's background has not been set using setBackground. |
| | setBackground | Sets the color used by update to erase the Component. |
| | getFont | Returns the Font used to render text in paint. Parent's Font is returned if font has not been set by setFont. |
| | setFont | Sets the Font used to display text in paint. |
| | getColorModel | Returns the color model used by the desktop to display the Component. |
| | getGraphics | Returns a Graphics object attached to this Component's on-screen rectangle. |
| | getFontMetrics | Returns a FontMetrics detailing the Component's Font (what is returned from getFont) as rendered to the Graphics display surface. |

## Component

**Purpose**

Abstracts all window components. Functionalities common to all such components are implemented in the Component class.

**Syntax**

public abstract class Component implements ImageObserver;

**Description**

An abstract windowing component. All Components have a bounding rectangle of on-screen space in which to render themselves. Java has several predefined

Component classes, such as a TextField, a Checkbox, or a Frame. The Component class implements methods for Event handling and management of an on-screen bounding rectangle. Figure 2-5 is an inheritance diagram for the Component class.

**Figure 2-5** Inheritance diagram of the Component class

**PackageName**

*java.awt*

**Imports**

*java.io.PrintStream,java.awt.peer.ComponentPeer, java.awt.image.ImageObserver, java.awt.image.ImageProducer, java.awt.image.ColorModel*

**Constructors**

None.

**Parameters**

None.

## action

**ClassName**

Component

**Purpose**

Event handler for ACTION_EVENT Events.

**Syntax**

public boolean action(Event evt, Object arg);

**Parameters**

***Event evt***

The ACTION_EVENT Event object.

***Object arg***

Argument attached to the Event object *evt*. This is identical to the *arg* member of *evt*.

**Description**

Called by the default implementation of handleEvent whenever an ACTION_EVENT is sent to the Component object. Action events include selection of a menu item and pressing a button. Override the default implementation of this method to make your Component react to action events.

**Imports**

None.

**Returns**

Returns true if the action event is handled by this Component object. A return value of false causes the Event to be automatically sent to the parent Container of this Component. The default implementation simply returns false.

**See Also**

The handleEvent method of the Component class

**Example**

The following example alternatively disables and enables a Go button whenever the Example button is pressed. The action method's *arg* in this case is the String title of the button that was pressed.

```
public class MyContainer extends Panel {
    Button buttonGo = new Button("Go");
    public MyContainer() {
        add(buttonGo);
        add(new Button("Example"));
    }

    action(Event evt, Object arg) {
        if(!(arg instanceof String))
            return false;

        if(((String)arg).equals("Example"))
            if(buttonGo.isEnabled())
                buttonGo.disable();
            else
                buttonGo.enable();
    }
}
```

## bounds

**ClassName**

Component

**Purpose**

Gets the bounding rectangle for this Component.

**Syntax**

public Rectangle bounds();

**Parameters**

None.

**Description**

This method gets a Rectangle object whose *x* and *y* members are set to the coordinates of the upper-left corner of the Component, relative to the origin of the parent Container. The *width* and *height* members of the Rectangle are set to the dimensions of the Component.

**Imports**

None.

**Returns**

A Rectangle object describing the bounding rectangle of this Component object is returned. The values expressed are relative to the origin of the parent Container object. The *x* and *y* members describe the upper-left corner of the Component.

**See Also**

The java.awt.Rectangle class

**Example**

The following example calculates and writes out the exact coordinates of the lower-right corner of the Component rectangle.

```
public class MyComponent extends Component {
```

```
    ...

    public void outputLowerRightCoords() {
        Rectangle r = bounds();
        System.out.println( new Point( r.x + r.width,
                                       r.y + r.height ) );
        return;
    }

    ...
}
```

## checkImage

**ClassName**

Component

**Purpose**

To check the status of construction of an Image.

**Syntax**

public int checkImage(Image img, ImageObserver observer); public int checkImage(Image img, int width, int height, ImageObserver observer);

**Parameters**

***Image img***

The Image object whose status is to be checked.

***int width***

The scaled size of the image representation being checked.

***int height***

***ImageObserver observer***

An ImageObserver object currently being notified of the progress of construction of the Image object.

**Description**

Checks the status of the construction of an Image object. The second overloaded version of this method checks the construction of a scaled representation of the Image object.

**Imports**

None.

**Returns**

A logical ORing of the ImageObserver flags indicating what information about the Image is available. This can include one or more of the following ImageObserver values: WIDTH, HEIGHT, PROPERTIES, SOMEBITS, FRAMEBITS, ALLBITS, ERROR.

**See Also**

The ImageObserver interface

**Example**

This example prevents the Component from painting its surface until the Image construction flag ALLBITS has been passed to the ImageObserver watching the image construction process.

```
public MyComponent extends Canvas {
    Image _img;
```

```
    // Constructor takes an Image parameter and begins
    // construction of it.
    public MyComponent(Image img) {
        _img = img;
        prepareImage(_img, this); // using this Component
                                  // as the ImageObserver.
    }
    // paint does nothing until image has been
    // fully constructed.
    public void paint(Graphics g) {
        if(0 == (ImageObserver.ALLBITS &
                 checkImage(_img, this)))
            return;

        // Do something with the image
        ...
    }
}
```

## createImage

**ClassName**

Component

**Purpose**

Creates an in-memory Image with a specified width and height, or from the output of an ImageProducer object.

**Syntax**

public Image createImage(int *x*, int *y*); public Image createImage(ImageProducer producer);

**Parameters**

***int x***

The width and height of the resultant Image object.

***int y***

***ImageProducer producer***

The ImageProducer object which will provide the data that defines the resultant Image.

**Description**

Creates an Image object of the specified width and height. This Image is suitable for drawing on for double-buffered screen updating. (See Chapter 1's discussion of double-buffered updating.) The resultant Image will have a compatible ColorModel to the display device associated with this Component object. The second overloaded version of this method creates the Image using data from the ImageProducer.

**Imports**

*java.awt.image.ImageProducer*

**Returns**

An Image object. The Image has not been constructed yet. Use Component.prepareImage() to begin construction of a screen representation of the Image.

**See Also**

The ImageProducer class

**Example**

This example creates an in-memory Image, draws on it, and then renders the entire in-memory Image to the Component's display surface.

```
public void paint(Graphics g) {
    Image imgTemp = createImage(size().width,
            size().height);
    Graphics gTemp = imgTemp.getGraphics();
    for( int ii=0 ; ii<10 ; ii++ )
        gTemp.drawLine(0, 0, size().width / ii,
                size().height);
    g.drawImage(imgTemp, 0, 0, this); // Use this Component
                                      // as the ImageObserver.
}
```

## deliverEvent

**ClassName**

Component

**Purpose**

Called within your Java code to deliver an Event to any Component object. The Java system uses a different mechanism to deliver user-generated Events.

**Syntax**

public void deliverEvent(Event evt);

**Parameters**

***Event evt***

The Event object to deliver to this Component.

**Description**

Delivers an Event object to this Component. The default implementation simply calls postEvent. To send a custom Event to a Component object, use the Component's deliverEvent method. This ensures the Event will automatically be routed to the Component's Container if the Component does not handle the Event. The Java system uses a different mechanism to deliver user-generated Events to a Component. That mechanism involves calling the Component's postEvent method directly using a special callback Thread, without using deliverEvent.

**Imports**

*java.awt.Event*

**Returns**

None.

**See Also**

The Event class

**Example**

This example delivers a custom Event object to a Component. The Component has been given a custom handleEvent to handle the custom Event type.

```
public class MyEvent extends Event {
    public static final MY_EVENT_ID = 5000; // any value
    public MyEvent(Component target, Object arg) {
        super(target, MY_EVENT_ID, arg);
    }
}

public class MyClass {
    Component _c;

    public MyClass(Component c) {
        _c = c;
    }

    public deliverEvent(Object arg) {
        c.deliverEvent(new MyEvent(c, arg));
    }
}
```

## disable

**ClassName**

Component

**Purpose**

Disables a Component, which prevents delivery of user-interaction Events to the Component.

**Syntax**

public void disable();

**Parameters**

None.

**Imports**

None.

**Description**

Disables the component. The Component's peer is also disabled as a result of calling this method. Predefined Components, such as Buttons or Labels, take on a grayed outlook when they are disabled. All Components, either predefined or custom ones, no longer receive user-interaction events once they are disabled.

**Returns**

None.

**Example**

See the example for the action method of the Component class.

## enable

**ClassName**

Component

**Purpose**

Enables or disables the Component.

**Syntax**

public void enable(); public void enable(boolean fEnabled);

**Parameters**

***boolean fEnabled***

If true, the Component is enabled. If false, the Component is disabled.

**Imports**

None.

**Description**

Enables the Component. The Component's peer is also enabled as a result of calling this method. The second overloaded version will enable or disable the Component according to the boolean value passed. A disabled Component no longer receives user-generated Events, such as mouse or keyboard Events. Predefined Components, such as Buttons or Labels, take on a grayed out look to denote to the user that they are disabled. Enabled Components receive all user-generated Events.

**Returns**

None.

**Example**

See the example for the action method of the Component class.

## getBackground

**ClassName**

Component

**Purpose**

Gets this Component's current background color.

**Syntax**

public Color getBackground()

**Parameters**

None.

**Imports**

*java.awt.Color*

**Description**

Gets the background Color object that is automatically applied to Graphics objects passed to the paint method.

**See Also**

The Color class; the getForeground, setForeground, and setBackground methods of the Component class

**Example**

This example code snippet demonstrates the default implementation of the update method of the Component class. The only reference to a Component's background color within the Java API is within update, which uses the background color to erase the Component's entire display surface.

```
public void update(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(0, 0, size().width, size().height);
    g.setColor(getForeground());
    paint(g);
```

```
}
```

## getColorModel

**ClassName**

Component

**Purpose**

Gets the ColorModel for the display surface attached to this Component object.

**Syntax**

public ColorModel getColorModel();

**Imports**

*java.awt.ColorModel*

**Description**

Gets the ColorModel for the display surface attached to this Component object. A ColorModel provides methods for converting pixel values to red, green, blue, and alpha color component values. In Java 1.0, this method returns an IndexedColorModel, from which you can get the current palette for the system's desktop.

**Returns**

A ColorModel that encapsulates methods for converting pixel values to red, green,blue, and alpha color components when displayed on the Component's display surface.

**See Also**

The ColorModel class

**Example**

This example displays the number of bits/pixel for the Component's display surface.

```
public MyApplet extends Applet {
    Button buttonEx = new Button("Example");

    public MyApplet() {}

    public void init() {
        add(buttonEx);
    }

    public void start() {
        System.out.println(buttonEx.getColorModel().getPixelSize());
    }
}
```

## getFont

**ClassName**

Component

**Purpose**

Gets the Font object associated with this Component.

**Syntax**

public Font getFont();

**Imports**

*java.awt.Font*

**Description**

Gets the Font associated with this Component. You associate a Font with a Component using setFont. Note that the Font returned by getFont may still have to be selected by the Component's Graphics object using Graphics.setFont. For the predefined Component classes in the *java.awt* package, such as Button and List, it is not necessary for the Graphics object to select the Font. But for custom Components, you will have to add a line like this to your paint method to ensure the Component's Font is selected into the Graphics object:

```
public void paint(Graphics g) {
    g.setFont(getFont());
    ...
}
```

**Returns**

The Font object currently associated with this Component. If a Font has not been associated with this Component, using setFont will get the parent Container's Font.

**See Also**

The Font class; the setFont method of the Component class

**Example**

This example method bolds a Component's Font when called.

```
public void makeFontBold(Component c) {
    Font f = c.getFont();
    c.setFont(new Font(f.getName(),
            f.getStyle() | Font.BOLD,
            f.getSize()));
}
```

## getFontMetrics

**ClassName**

Component

**Purpose**

Gets the FontMetrics for a specified Font as it is rendered on the Component's display surface.

**Syntax**

public FontMetrics getFontMetrics(*Font f*);

**Parameters**

***Font f***

The Font for which to create the FontMetrics.

**Imports**

*java.awt.Font, java.awt.FontMetrics*

**Description**

Gets the FontMetrics for the passed Font. The FontMetrics are for the display surface associated with this Component. You can use the return value from getFont as the Font parameter to this method, like so

```
FontMetrics fm = getFontMetrics(getFont());
```

**See Also**

The Font class and the FontMetrics class

**Example**

This example measures the width in pixels of a given string on the Component's display surface using the Component's Font.

```
public class MyComponent extends Canvas {
    public int measureString(String str) {
        return getFontMetrics(getFont()).stringWidth(str);
    }
}
```

## getForeground

**ClassName**

Component

**Purpose**

Gets the Color used for foreground painting on the Component's display surface in the paint method.

**Syntax**

public Color getForeground();

**Parameters**

None.

**Imports**

*java.awt.Color*

**Description**

Gets the foreground Color for this Component. The Foreground color is automatically associated with Graphics objects passed to the paint method by the default implementation of update. See the example of the getBackground method to see how this association happens.

**Returns**

A Color object representing the current color of the Graphics object passed to paint.

**See Also**

The Color class; the setForeground method of the Component class

**Example**

This example sets the foreground and background colors of an in-memory Image object's Graphics to be the same as the component's foreground and background colors.

```
public class MyComponent extends Canvas {
    ...

    public Graphics makeImageHaveCompatibleColors(Image img) {
        Graphics g = img.getGraphics();
        g.setColor(getBackground());
        g.fillRect(0, 0, size().width, size().height);
```

```
        g.setColor(getForeground());
        return g;
    }

    ...
}
```

## getGraphics

**ClassName**
Component
**Purpose**
Gets a Graphics object whose display surface is the rectangle of the on-screen desktop controlled by this Component.
**Syntax**
public Graphics getGraphics();
**Parameters**
None.
**Imports**
*java.awt.Graphics*
**Description**
Returns a Graphics object for this Component. If the Component has not been added to a Container using Container.add, this method will return null. The foreground Color, background Color, and Font have been selected by the Graphics object.
**Returns**
A Graphics object attached to the on-screen rectangle controlled by this Component.
**See Also**
The Graphics class
**Example**
See the example for the getForeground method of the Component class.

## getParent

**ClassName**
Component
**Purpose**
Gets the parent Container of this Component.
**Syntax**
public Container getParent();
**Parameters**
None.
**Imports**
*java.awt.Container*

**Description**

Gets the Container for this Component object. If the Component is a top-level Frame window, or the Component has not been added to a Container using Container.add, then this method will return null.

**Returns**

A reference to the Container object which controls this Component. If the Component does not have a parent Container, null will be returned.

**See Also**

The add method of the Container class

**Example**

This example places the Component in the lower-right corner of the parent Container's bounding rectangle.

```
public class MyComponent extends Component {
    ...

    public void placeParentsLowerRight() {
        Container parent = getParent();
        Dimension dimParent = parent.size();
        Dimension dimThis = size();
        move(rectParent.width - dimThis.width,
                rectParent.height - dimThis.height);
    }

    ...
}
```

## getPeer

**ClassName**

Component

**Purpose**

Gets the ComponentPeer associated with this Component object. The ComponentPeer is the proxy through which calls to the native windowing system are made.

**Syntax**

public ComponentPeer getPeer();

**Parameters**

None.

**Imports**

*java.awt.peer.ComponentPeer*

**Description**

Gets the Peer object associated with this Component. If this Component has no Peer, null will be returned.

**Returns**

A Reference to the ComponentPeer attached to this Component. If no ComponentPeer exists, as would be the case if the Component has not been added to a Container using Container.add, then null will be returned.

**See Also**

The ComponentPeer class

**Example**

This example implements a *virtual Component*, which is a Component that does not have a peer. Virtual Components are useful because they inherit all the Component bounding rectangle and Event handling methods. Virtual Components can be used to manage overlapping rectangles of on-screen space, especially because sibling virtual Components will not "clip" each other on the desktop.

```
public class VirtualComponent extends Canvas {
    ...

    // Overriding addNotify ensures no ComponentPeer
    // will ever be created for this object.
    public void addNotify() {
        return;
    }

    ...
}
```

## getToolkit

**ClassName**

Component

**Purpose**

Gets the Toolkit object, which is the proxy for the native windowing system itself.

**Syntax**

public Toolkit getToolkit()

**Imports**

*java.awt.Toolkit*

**Description**

Gets the Toolkit object for this Java session. The Toolkit is the proxy for the native windowing system on the local computer. Through the Toolkit you can create ComponentPeers and retrieve various windowing system parameters such as the list of available Fonts.

**Returns**

A reference to the Toolkit associated with this Component.

**See Also**

The Toolkit class

**Example**

The Toolkit can be used to download images or audio clips directly. This example uses a Component's Toolkit to download an Image from within a non-Component object.

```
public class MyClass {
    Frame _frame = new Frame("Sample");
    Image _img;

    public MyClass() {
        _frame.show();

        Toolkit tk = _frame.getToolkit();
        _img = tk.getImage(
```

```
                new URL("http://www.co.com/logo.gif"));
    }
}
```

## gotFocus

**ClassName**

Component

**Purpose**

Event handler method for GOT_FOCUS Events.

**Syntax**

public boolean gotFocus(Event evt, Object arg);

**Parameters**

***Event evt***

The GOT_FOCUS Event sent to this Component.

***Object arg***

The argument to the GOT_FOCUS Event. This parameter is identical to the *arg* member of *evt*.

**Imports**

***java.awt.Event***

**Description**

This notification method is called by the default implementation of handleEvent when a GOT_FOCUS Event is sent to this Component, indicating that this Component has the keyboard focus. The default implementation of the Event handling method simply returns false. Override the default implementation to allow your Component to react when the Component receives the keyboard focus.

**Returns**

Have your default implementation return true, indicating the GOT_FOCUS Event has been handled. If false is returned, then the Event will be posted to the this Component's parent Container.

**Example**

This example reports to System.out when the Component receives or loses keyboard focus.

```
public class MyComponent extends Canvas {
    ...

    public boolean gotFocus(Event evt, Object arg) {
        System.out.println(arg + " got keyboard focus.");
        return true;
    }

    public boolean lostFocus(Event evt, Object arg) {
        System.out.println(arg + " lost keyboard focus.");
        return true;
    }
}
```

## handleEvent

**ClassName**

Component

**Purpose**

Called to allow the Component a chance to handle user-generated or other Events.

**Syntax**

public boolean handleEvent(Event evt);

**Parameters**

***Event evt***

The Event to be handled by this Component.

**Imports**

*java.awt.Event*

**Description**

This method acts as a central clearing house for all events sent to this Component, or unhandled events sent to subcomponents of this object. The default implementation is a large switch() statement which calls more specific methods, such as keyDown(), mouseMove(), gotFocus(), etc. The return value indicates whether the Event has been handled or should be sent to the parent Container.

**Returns**

A return value of true indicates the Event has been handled. False indicates it has not, and the Event will be sent to the parent Container.

**See Also**

All of the Event handle methods of the Component class; the Event class

**Example**

In this example, it is known that the defined class of Components never handles any Events. The default implementation of handleEvent will still run through its long switch statement and attempt to find a handler for the Event. This class is optimized to stop the Java system from performing that unnecessary handleEvent code.

```
public class MyNoHandlerComponent extends Component {
    ...

    public boolean handleEvent(Event evt) {
        return false;
    }

    ...
}
```

## hide

**ClassName**

Component

**Purpose**

Makes the Component invisible or “hidden.”

**Syntax**

public void hide();

**Parameters**

None.

**Imports**

None.

**Description**

Hides the Component. Hidden components are not drawn, nor do they take up space on the display surface.

**Returns**

None.

**See Also**

The show method of the Component class

**Example**

This example Panel uses a Label to display a countdown. When the countdown reaches 0, the Label is hidden. Note that this example does not halt its background Thread in its stop() method implementation for purposes of readablility.

```
public class MyCountDownApplet extends Applet implements
        Runnable {
    Label _label = new Label();
    int _nCount = 10;

    public MyCountDownApplet() {}

    public void start() {
        add(_label);
        show();
        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        while(_nCount >= 0 ) {
            try {
                Thread.currentThread().sleep(1000);
            } catch (Exception e) {}
            _nCount--;
            _label.setText(new String(""+_nCount));
        }

        _label.hide();
    }
}
```

## imageUpdate

**ClassName**

Component

**Purpose**

This is the only method of the ImageObserver interface. The default implementation repaints the entire Component whenever any progress is made in the construction of an image.

**Syntax**

public boolean imageUpdate(Image img, int flags, int x, int y, int width, int height);

**Parameters**

***Image img***

The Image object to check. If progress on a screen representation of this Image has been made, then the Component will be repainted asynchronously.

***int flags***

The ImageObserver flags indicating the progress of construction of a screen representation of the Image object. These ImageObserver flags are ORed together.

***int x***

Indicates the rectangle of the Image for which the *flags* parameter is valid.

***int y***

***int width***

***int height***

**Imports**

None.

**Description**

Causes an asynchonous repainting of the Component if construction of the Image's representation has made progress. The same *flags* parameter as is returned by checkImage method is passed. Because this method is implemented in the Component class, any Component object may act as an ImageObserver.

**Returns**

Returns true if further notification of image construction should continue. False causes further notification to be terminated.

**See Also**

The imageUpdate method of the ImageObserver interface

**Example**

This example prevents the Component from repainting unless the ALLBIT flag is passed as part of the *flags* parameter.

```
public class MyComponent extends Canvas {
    ...

    public boolean imageUpdate(Image img, int flags, int x,
            int y, int width, int height) {
        if(0 != (flags & ImageObserver.ALLBITS))
            return true;

        repaint();
        return false;
    }

    ...
}
```

## inside

**ClassName**

Component

**Purpose**

Checks to see if a particular point lies within this Component's bounding rectangle.

**Syntax**

public boolean inside(int *x*, int *y*);

**Parameters**

***int x***

The coordinates of the point to check.

***int y***

**Imports**

None.

**Description**

Checks whether a particular point lies inside or outside the Component's bounding rectangle. The point to check is specified relative to the parent Container's origin (generally its upper-left corner).

**Returns**

True is returned if the point lies within this Component's bounding rectangle. Otherwise false is returned.

**Example**

This example method moves the Component's origin to a point if that point does not lie within the Component's bounding rectangle.

```
public class MyComponent extends Canvas {
    ...

    public void moveOver(int x, int y) {
        if(inside(x, y))
            return;

        move(x, y);
    }

    ...
}
```

## invalidate

**ClassName**

Component

**Purpose**

Marks the Component as invalid. Calls to Component.validate are ignored unless the Component has been marked as invalid.

**Syntax**

public void invalidate();

**Parameters**

None.

**Imports**

None.

**Description**

Sets an internal boolean variable, indicating the Component must be validated. Use validate to re-validate the Component Container components. In conjunction with a LayoutManager object, use the invalidate/validate methods to layout

subcomponents. By default implementation, Component objects do not react to being tagged as invalid.

**Returns**

None.

**See Also**

The Container class and the LayoutManager interface

## isEnabled

**ClassName**

Component

**Purpose**

Tells whether or not the Component is enabled.

**Syntax**

public boolean isEnabled();

**Parameters**

None.

**Imports**

None.

**Description**

Checks to see if the Component is currently enabled. When created, a Component is enabled. The disable method is used to disable a Component.

**Returns**

True if the Component is currently enabled. False if it is not.

**Example**

The paint method of this Component draws differently if the Component is disabled.

```
public class MyComponent extends Canvas {$    ...

    public void paint(Graphics g) {
        if(isEnabled()) {
            // Draw enabled version of the Component.
        } else {
            // Draw disabled version of the Component.
        }
    }

    ...
}
```

## isShowing

**ClassName**

Component

**Purpose**

Tells whether or not any part of the Component is currently showing on the desktop.

**Syntax**

public boolean isShowing();

**Parameters**

None.

**Imports**

None.

**Description**

Checks to see whether the Component object is currently showing on the display screen. The Component is not showing if it is currently hidden. It is not showing if its bounding rectangle has a 0 dimension along either axis. It is not showing if the intersection of its bounding rectangle with its parent's bounding rectangle has a 0 dimension along either axis. That is, Component positioned outside the bounds of its parent Container's bounding rectangle.

**Returns**

True is returned by this method for all Components which currently have some rectangle of screen real estate. False is returned otherwise.

**See Also**

The hide and show methods of the Component class

**Example**

This Component is optimized by immediately returning from its paint implementation if it is not currently showing.

```
public class MyComponent extends Canvas {
    ...

    public void paint(Graphics g) {
        if(!isShowing())
            return;

        // Draw the Component...
    }

    ...
}
```

## isValid

**ClassName**

Component

**Purpose**

Tells whether or not this Component is currently flagged as "nvlaid.

**Syntax**

public boolean isValid();

**Parameters**

None.

**Imports**

None.

**Description**

Checks to see whether the Component object is currently valid. Each Component has an internal invalid flag. The invalidate method is used to set this flag, and the

validate method is used to clear the flag. The default implementation of validate actually does nothing except clear the flag. The Container class, however, uses the invalid flag and the validate method as an indication of when it should rearrange its child Components with the help of its LayoutManager.

**Returns**

Valid Components return true from calls to this method. Invalid ones return false.

**See Also**

The invalidate and validate methods of the Component class

## isVisible

**ClassName**

Component

**Purpose**

Tells whether or not the Component is currently visible.

**Syntax**

public boolean isVisible();

**Parameters**

None.

**Imports**

None.

**Description**

Checks to see whether the Component object is currently hidden. To hide a Component object, call its hide method. The show method will alternatively make the object unhidden. Components, when created, are not hidden by default.

**Returns**

If the Component is currently hidden, false is returned. If it is not hidden, true is returned.

**Example**

This Component object suspends its background processing Thread while it is hidden. This is accomplished by overriding its show method and by implementing the Runnable interface.

```
public class MyComponent extends Canvas implements
        Runnable {
    Thread _t = new Thread(this);

    public MyComponent() {
        _t.start();
    }

    public synchronized void quit() {
        Thread tTemp = _t;
        _t = null;
        notify();
    }
}

    public synchronized void show() {
        notify();
        super.show();
```

```
    }

    public synchronized void run() {
        while(null != _t) {
            if(!isVisible()) {
                wait();
                continue;
            }

            // do one iteration of background processing
        }
    }

    ...
}
```

## keyDown

**ClassName**

Component

**Purpose**

Event handler for KEY_PRESS Events.

**Syntax**

public boolean keyDown(Event evt, int key);

**Parameters**

***Event evt***

The KEY_PRESS or KEY_ACTION Event which was sent to this Component.

***int key***

The key pressed. Note that the key is also stored in the *key* member of *evt*.

**Imports**

*java.awt.Event*

**Description**

This method is called by the the default implementation of handleEvent whenever a KEY_PRESS or KEY_ACTION Event is sent to the Component. The passed parameters indicate the code for the key pressed. A custom implementation should return true if the event is handled by the Component and should not be sent on to the Component's Container.

**Returns**

The default implementation of this method simply returns false, indicating the Event should be passed on to the parent Container's handleEvent method.

**See Also**

The Event class; the handleEvent method of the Component class

**Example**

This example Component changes its background color whenever the spacebar is pressed. The Component is not repainted until a KEY_RELEASE Event is sent to the Component.

```
public class MyComponent extends Canvas {
    static Color[] ac = new Color[2];
    int i = 0;
```

```
    public MyComponent() {
        ac[0] = new Color(0, 0, 0);
        ac[1] = new Color(255, 255, 255);
    }

    public boolean keyDown(Event evt, int key) {
        if(' ' = (char)key) {
            i++;
            i %= 2;
            setBackground(ac[i]);
            return true;
        }

       return false;
    }

    public boolean keyUp(Event evt, int key) {
        if(' ' == (char)key) {
            repaint();
            return true;
        }

        return false;
    }
}
```

## keyUp

**ClassName**

Component

**Purpose**

Event handler for KEY_RELEASE Events.

**Syntax**

public boolean keyUp(Event evt, int key);

**Parameters**

***Event evt***

The KEY_RELEASE or KEY_ACTION_RELEASE Event which was sent to this Component.

***int key***

The key pressed. Note that the key is also stored in the *key* member of *evt*.

**Imports**

None.

**Description**

This method is called by the the default implementation of handleEvent whenever a KEY_RELEASE or KEY_ACTION_RELEASE Event is sent to the Component. The passed parameters indicate the code for the key pressed. A custom implementation should return true if the event is handled by the Component and should not be sent on to the Component's Container.

**Returns**

The default implementation of this method simply returns false, indicating the Event should be passed on to the parent Container's handleEvent method.

**See Also**

The Event class; the handleEvent method of the Component class

**Example**

See the example for the keyDown method of the Component class.

## layout

**ClassName**

Component

**Purpose**

Called by the default implementation of validate if the Component is currently invalid. The default implementation of Component.layout does nothing.

**Syntax**

public void layout();

**Parameters**

None.

**Imports**

None.

**Description**

Called when invalid Component objects are being validated as part of validate. This method is primarily used to layout child Components in Container objects. The default implementation of Component.layout does nothing. The Container class implementation of layout relies on a LayoutManager object to handle the laying out of child Components.

**Returns**

None.

**See Also**

The validate and invalidate methods of the Component class

## list

**ClassName**

Component

**Purpose**

To display the internal state of the Component to a PrintStream object.

**Syntax**

public void list();
public void list(PrintStream out);
public void print(PrintStream out, int indent);

**Parameters**

***PrintStream out***

The stream to write a textual description of the state of this Component to.

***int indent***

The number of space characters (ASCII char 32) to prepend to each line of text written to out.

**Imports**

*java.io.PrintStream*

**Description**

Outputs a textual description of the internal state of the Component to a PrintStream. This can be useful for debugging purposes. The first overloaded version of this method writes the listing to System.out, with an indentation of 0. A call to list() (without parameters) is equivalent to System.out.println(this).

**See Also**

The PrintStream class; the toString method of the Component class

**Example**

This example Container class method displays the Container's entire subtree of Components by tracing its hierarchy depth-first, indicating depth by indentation on the PrintStream. Output is written to System.out.

```
public class MyContainer extends Panel {
    ...

    public void displayChildren(Container target) {
        displayChildren(target, 0);
    }

    public void displayChildren(Container target, int indent) {
        Component[] acomps =
                new Component[target.countComponents()];

        for( int ii=0 ; ii<acomps.length ; ii++ ) {
            acomps[ii].list(System.out, indent);
            if(acomps[ii] instanceof Container)
                displayChildren((Component)
                        acomps[ii], indent+1);
        }
    }
}
```

## locate

**ClassName**

Component

**Purpose**

Returns a reference to this Component if the passed point lies within this Component's bounding rectangle.

**Syntax**

public Component layout(int x, int y);

**Parameters**

***int x***

***int y***

These two parameters describe a point, relative to this Component's origin, to test.

**Imports**

None.

**Description**

A hit-test method which checks to see which Component, or subcomponent, contains the point described by the passed *x* and *y* parameters. The Container class uses this method to determine which of its child Component's contains a particular point.

**Returns**

The Component, or subcomponent, that contains the point (*x*,*y*). If the point lies outside the bounds of this Component, null is returned. Container.locate() re-implements this method to test all subcomponents.

**See Also**

The Container class

## location

**ClassName**

Component

**Purpose**

Gets the location of this Component's origin.

**Syntax**

public Point location();

**Parameters**

None.

**Imports**

None.

**Description**

Gets the coordinates of the upper-left corner of this Component. The returned Point is relative to the parent Container's origin.

**Returns**

The coordinates of the upper-left corner of this Component.

**See Also**

The Point class

## lostFocus

**ClassName**

Component

**Purpose**

Event handler for LOST_FOCUS Events.

**Syntax**

public boolean lostFocus(Event evt, Object arg);

**Parameters**

***Event evt***

The LOST_FOCUS Event sent to this Component.

***Object arg***

The argument to the LOST_FOCUS Event. This parameter is identical to the *arg* member of *evt*.

**Imports**

*java.awt.Event*

**Description**

This notification method is called by the default implementation of handleEvent when a LOST_FOCUS Event is sent to this Component, indicating this Component no longer has the keyboard focus.

**Returns**

A return value of true indicates the LOST_FOCUS Event is handled by this Component. Returning false causes the event to automatically be sent to this Component's parent Container object.

**See Also**

The handleEvent method of the Component class

**Example**

See the example for the gotFocus method of the Component class.

## minimumSize

**ClassName**

Component

**Purpose**

Allows a Component to tell its parent Container the minimum bounding rectangle it requires.

**Syntax**

public Dimension minimumSize();

**Imports**

None.

**Description**

Returns the minimum sized rectangle of display surface required for this Component to display itself. When the Component's Container lays out its subcomponents, this method is called to establish a minimum amount of screen real estate needed by the Component.

**Returns**

The returned Dimension object should indicate the minimum required width and height needed for this Component to display itself. The default implementation of Component.minimumSize() returns the minimum size required as indicated by the Component's peer. If no peer exists, the current size of the Component is returned.

**See Also**

The preferredSize method of the Component class; the LayoutManager interface; the Dimension class

**Example**

This example custom Component reports that it requires at least a 10 x 10 bounding rectangle, but would prefer a bounding rectangle large enough to display an initializer String.

```
public class MyComponent extends Canvas {
    String _str;

    public MyComponent(String str) {
        _str = str;
    }

    public Dimension minimumSize() {
        return new Dimension(10, 10);
    }

    public Dimension preferredSize() {
        FontMetrics fm = getFontMetrics(getFont());
        return new Dimension(fm.getHeight(),
                fm.stringWidth(_str));
    }

    ...
}
```

## mouseDown

**ClassName**

Component

**Purpose**

Event handler for MOUSE_DOWN Events.

**Syntax**

public boolean mouseDown(Event evt, int x, int y);

**Parameters**

***Event evt***

The MOUSE_DOWN Event sent to this Component.

***int x***

***int y***

The on-screen coordinates where the mouse was clicked. The coordinates are expressed relative to the origin of this Component object. These two parameters are identical to the *x* and *y* members of *evt*.

**Imports**

*java.awt.Event*

**Description**

This notification method is called by the default implementation of handleEvent when a MOUSE_DOWN Event is sent to this Component, indicating the user has clicked the mouse inside this Component.

**Returns**

A return value of true indicates the MOUSE_DOWN Event is handled by this Component. Returning false causes the event to automatically be sent to this Component's parent Container object.

**See Also**

The handleEvent, mouseUp, and mouseDrag methods of the Component class

**Example**

This simple example custom Component prints the coordinates of the mouse while the mouse button is down.

```
public class MyComponent extends Component {
    boolean _bMouseIsDown = false;
    Point _ptMouseCoords = null;

    public MyComponent() {}

    public void paint(Graphics g) {
        if((null == _ptMouseCoords) ||
                (false == _bMouseIsDown))
            return;

        g.drawString("”+_ptMouseCoords, _ptMouseCoords.x,
                _ptMouseCoords.y);
    }

    public boolean mouseDown(Event evt, int x, int y) {
        _ptMouseCoords = new Point( x, y);
        _bMouseIsDown = true;
        repaint();
    }

    public boolean mouseDrag(Event evt, int x, int y) {
        _ptMouseCoords = new Point(x, y);
        _bMouseIsDown = true;
        repaint();
    }

    public boolean mouseUp(Event evt, int x, int y) {
        _bMouseIsDown = false;
        repaint();
    }
}
```

## mouseDrag

**ClassName**

Component

**Purpose**

Event handler for MOUSE_DRAG Events.

**Syntax**

public boolean mouseDrag(Event evt, int x, int y);

**Parameters**

***Event evt***

The MOUSE_DRAG Event sent to this Component.

***int x***

***int y***

The coordinates where the mouse was dragged to. These parameters are identical to the *x* and *y* members of *evt*.

**Imports**

*java.awt.Event*

**Description**

This notification method is called by the default implementation of handleEvent when a MOUSE_DRAG Event is sent to this Component, indicating that the mouse has been moved while the mouse button is held down.

**Returns**

A return value of true indicates the MOUSE_DRAG Event is handled by this Component. Returning false causes the event to automatically be sent to this Component's parent Container object.

**See Also**

The handleEvent, mouseDown, mouseUp, and mouseMove methods of the Component class; the Event class

**Example**

See the example for the mouseDown method of the Component class.

## mouseEnter

**ClassName**

Component

**Purpose**

Event handler for MOUSE_ENTER Events.

**Syntax**

public boolean mouseEnter(Event evt, int x, int y);

**Parameters**

***Event evt***

The MOUSE_ENTER Event sent to this Component.

***int x***

***int y***

The argument to the GOT_FOCUS Event. This parameter is identical to the *arg* member of *evt*.

**Imports**

*java.awt.Event*

**Description**

This notification method is called by the default implementation of handleEvent when a MOUSE_ENTER Event is sent to this Component, indicating that the mouse cursor has moved onto this Component's display rectangle. A Component will only receive a single MOUSE_ENTER Event before a subsequent MOUSE_EXIT Event is sent. That is, each call to mouseEnter is matched by exactly one subsequent call to mouseExit.

**Returns**

A return value of true indicates the MOUSE_ENTER Event is handled by this Component. Returning false causes the event to automatically be sent to this Component's parent Container object.

**See Also**

The handleEvent and mouseExit methods of the Component class; the Event class

**Example**

This example component uses the mouseEnter and mouseExit Event handlers to detect when the mouse cursor is over it. When the mouse is over it, the Component exchanges its background and foreground colors compared to when the mouse is not over it.

```
public class MyComponent extends Canvas {
    ...

    public boolean mouseEnter(Event evt, int x, int y) {
        Color bg = getBackground();
        setBackground( getForeground() );
        setForeground( bg );
        return true;
    }

    public boolean mouseExit(Event evt, int x, int y) {
        Color bg = getBackground();
        setBackground( getForeground() );
        setForeground( bg );
        return true;
    }

    ...
}
```

## mouseExit

**ClassName**

Component

**Purpose**

Event handle for MOUSE_EXIT Events.

**Syntax**

public boolean mouseExit(Event evt, int x, int y);

**Parameters**

***Event evt***

The MOUSE_EXIT Event sent to this Component.

***int x***

***int y***

The coordinates of the first point outside the Component's bounds that the mouse is moved to after being inside the bounds. Coordinates are relative to the upper-left corner of the Component.

**Imports**

*java.awt.Event*

**Description**

This notification method is called by the default implementation of handleEvent when a MOUSE_EXIT Event is sent to this Component, indicating that the mouse cursor has moved out of this Component's display rectangle. A Component will only receive a single MOUSE_ENTER event before a subsequent MOUSE_EXIT

event is sent. That is, each single call to mouseEnter is matched by a single subsequent call to mouseExit. MOUSE_EXIT Events are still sent to a Component even if the user is dragging the mouse (that is, moving the mouse while the mouse button is held down).

**Returns**

A return value of true indicates the MOUSE_EXIT Event is handled by this Component. Returning false causes the event to automatically be sent to this Component's parent Container object.

**See Also**

The Event class; the handleEvent and mouseEnter methods of the Component class

**Example**

See the example for the mouseEnter method of the Component class.

*The MouseMove method is described at the end of this chapter on page 135.*

## mouseUp

**ClassName**

Component

**Purpose**

Event handler for MOUSE_UP Events.

**Syntax**

public boolean mouseUp(Event evt, int x, int y);

**Parameters**

***Event evt***

The MOUSE_UP Event sent to this Component.

***int x***

The coordinates of the mouse cursor when the mouse button was released.

***int y***

These parameters are identical to the *x* and *y* members of *evt*.

**Imports**

*java.awt.Event*

**Description**

This notification method is called by the default implementation of handleEvent when a MOUSE_UP Event is sent to this Component, indicating the user has let go of the mouse button.

**Returns**

A return value of true indicates the MOUSE_UP Event is handled by this Component. Returning false causes the event to automatically be sent to this Component's parent Container object.

**See Also**

The Event class; the handleEvent, mouseDown, mouseMove, and mouseDrag methods of the Component class

**Example**

See the example for the mouseDown method of the Component class.

## move

**ClassName**
Component
**Purpose**
Moves the entire Component within its parent Container.
**Syntax**
public void move(int x, int y);
**Parameters**
***int x***
***int y***
The new coordinate of the upper-left corner of the Component object. The coordinates are expressed relative to the upper-left corner of the parent Container.
**Imports**
None.
**Description**
Relocates the Component relative to the upper-left corner of the parent Container. The dimensions of the moved Component are preserved.
**Returns**
None.
**See Also**
The location method of the Component class
**Example**
See the example for the inside method of the Component class.

## nextFocus

**ClassName**
Component
**Purpose**
Moves the keyboard focus to the next Component within the same Container.
**Syntax**
public void nextFocus();
**Parameters**
None.
**Imports**
None.
**Description**
Calling this method, moves the keyboard focus to the next Component within the same Container that is eligible to receive keyboard focus. Calling this method for a Component which does not currently have keyboard focus is a no-op. Use of the

Component.requestFocus method instead of nextFocus is strongly encouraged. See the example of the requestFocus method.

**Returns**

None.

**See Also**

The requestFocus, gotFocus, and lostFocus methods of the Component class

## paint

**ClassName**

Component

**Purpose**

Called whenever the Java system determines the Component must repaint its surface.

**Syntax**

public void paint(Graphics g);

**Parameters**

***Graphics g***

A Graphics object which has been attached to the display surface for the Component, and whose clipping rectangle has been set to whole or part of the Component's bounding rectangle.

**Imports**

*java.awt.Graphics*

**Description**

This method is called whenever the Component should render itself on the display surface. The Graphics object passed to this method is attached to the display surface, and is clipped to the Component's bounding rectangle. Custom Component objects should override this method. The paint method can be called by Java at any time, such as when your Java application is covered up by another application running at the same time. When the other application is removed from on top of your Java application, a paint call will be issued for all visible Components.

There are no guarantees on the internal state of the Graphics object, except that the clipping rectangle will be set to a rectangle equal to or contained by the Component's bounding rectangle. If you do not override the Component's default update method, then the foreground and background colors are also guaranteed to be selected in the Graphics object. In general, it is always safe to select the foreground Color, background Color, Font, and other special drawing features into the Graphics object, just to be sure that the Graphics' internal state is as it is expected to be. Use the repaint method to force an asynchronous paint call to be issued for the Component.

**Returns**

None.

**See Also**

The Graphics class; the repaint method of the Component class

**Example**

This example Component re-implements both the paint and repaint methods. The re-implementation of the repaint method guarantees that the foreground Color and Font are selected into paint's Graphics parameter, and the Component's surface has not been erased at all. Re-implementing the update method, to not erase a Component's surface, is the technique usually used to avoid flicker in graphics-intensive applications.

```
public class MyComponent extends Canvas {
    ...

    public void update(Graphics g) {
        g.setColor(getForeground());
        g.setFonr(getFont());
        paint(g);
    }

    public void paint(Graphics g) {
        // Draw something to the Component's surface...
    }

    ...
}
```

## paintAll

**ClassName**

Component

**Purpose**

Paints the Component after calling validate.

**Syntax**

public void paintAll(Graphics g);

**Parameters**

***Graphics g***

A Graphics object, which has been attached to the display surface for the Component, and whose clipping rectangle has been set to whole or part of the Component's bounding rectangle.

**Imports**

*java.awt.Graphics*

**Description**

This method paints the Component after calling validate. Note that this method is usually used to force a Container to repaint itself and all its child Components.

**Returns**

None.

**See Also**

The Graphics class; the validate method of the Component class

## postEvent

**ClassName**

Component

**Purpose**

Routes an Event to its handler method.

**Syntax**

public boolean postEvent(Event evt);

**Parameters**

***Event evt***

The Event object being sent to this Component.

**Imports**

*java.awt.Event*

**Description**

This method handles delivering an Event to either a Component's peer, the Component's handleEvent method, or to the Component's parent Container postEvent method (in that order). Note that the Java system calls a Component's postEvent method to deliver all Events to the Component. When delivering your own Events to a Component, use the deliverEvent method. The recursive design of the postEvent method is used to pass unhandled Events up from Component to parent Container to parent Container until some Event handler method returns true. deliverEvent simple calls postEvent. postEvent will actually allow the peer's handleEvent method to have a first shot at the Event. If the peer's handleEvent returns true, indicating the Event has been handled, then the Component's own handleEvent implementation is never called. This is the reason that, say, Scrollbar objects (which are directly derived from Component) cannot handle a MOUSE_DOWN or MOUSE_UP Event. Instead, the Scrollbar's peer handles these types of Events and changes them into SCROLLBAR_* Events to be handled by the Scrollbar. If neither the peer's nor the Component's handleEvent method handles the Event, the parent Container's postEvent method is passed the Event.

**Returns**

A return value of true indicates the Event has been handled by either the peer, this Component itself, or the Component's parent Container.

**See Also**

The Event class; the deliverEvent and handleEvent methods in the Component class

**Example**

Under some circumstances, you may actually want the Component's parent Container to take the first shot at handling the Component's Events. In this example, postEvent is re-implemented to allow the parent Container first shot at all Events.

```
public class MyComponent extends Canvas {
    ...

    public boolean postEvent(Event evt) {
        if(false == getParent().postEvent(evt))
            return super.postEvent(evt);
        return true;
    }

    ...
```

```
}
```

## preferredSize

**ClassName**
Component
**Purpose**
Allows the Component to tell its parent Container its preferred amount of on-screen real estate.
**Syntax**
public Dimension preferredSize();
**Parameters**
None.
**Imports**
*java.awt.Dimension*
**Description**
Returns the preferred size of the rectangle of display surface for this Component to display itself. When the Component's Container lays out its subcomponents, this method is called to establish a preferred amount of screen real estate for the Component. Re-implement this method to request a particular preferred size for your custom Component.
**Returns**
The returned Dimension object should indicate the preferred width and height for this Component to display itself. The default implementation of preferredSize returns the preferred size as indicated by the Component's peer. If no peer exists, the current size of the Component is returned.
**See Also**
The Dimension class; the minimumSize method of the Component class
**Example**
See the example for the minimumSize method of the Component class.

## prepareImage

**ClassName**
Component
**Purpose**
Kick-starts the Image construction process.
**Syntax**
public boolean prepareImage(Image img, ImageObserver observer); public boolean prepareImage(Image img, int width, int height, ImageObserver observer);
**Parameters**
***Image img***

The Image object to create a screen representation of.

***int width***

***int height***

The scaled size of the Image's representation.

***ImageObserver observer***

The ImageObserver object that receives notification of the asynchronous progress of the construction of the Image's representation.

**Imports**

*java.awt.Image, java.awt.image.ImageObserver*

**Description**

Starts construction of a screen representation of an Image object. The second overloaded version begins construction of a scaled version of the Image. An Image must be constructed before it can be displayed on a Component's surface. Note, that when you use Graphics.drawImage with a reference to an unconstructed Image object, the Image's construction process is automatically started for you. The prepareImage method allows you to start this process before the Image is displayed on any surface.

**Returns**

True is returned if the representation of the Image object is complete. Otherwise, false is returned and the Image construction process is started.

**See Also**

The Image class; the ImageObserver interface; the checkImage and updateImage methods of the Component class

**Example**

See the example for the checkImage method of the Component class.

## print

**ClassName**

Component

**Purpose**

To render the Component to a printer device.

**Syntax**

public void print(Graphics g);

**Parameters**

***Graphics g***

A Graphics object that has been attached to a printer device, and whose clipping rectangle has been set to whole or part of the Component's bounding rectangle.

**Imports**

*java.awt.Graphics*

**Description**

This method is called whenever the Component should render itself on a printer device. The Graphics object passed to this method has been attached to a printing device, and it is clipped to the Component's bounding rectangle. The default implementation simply calls paint, using the same Graphics object. Override this method if your custom Component is to be displayed differently when printed

compared to on a display device. There are no guarantees on the internal state of the Graphics object, except that the clipping rectangle will be set to a rectangle equal to or contained by the Component's bounding rectangle. It is a good idea to select the foreground Color, background Color, Font, and other special drawing features into the Graphics object to be sure the Graphics' internal state is as it is expected to be. Note that the print method is essentially the same thing as the paint method. The print method is provided for those instances when you need to know that your Component is being rendered to a printer.

**Returns**

None.

**See Also**

The Graphics class; the paint method of the Component class

**Example**

See the example for the paint method. The paint method and the print method are essentially the same thing.

## printAll

**ClassName**

Component

**Purpose**

Prints the Component after calling validate.

**Syntax**

public void printAll(Graphics g);

**Parameters**

***Graphics g***

A Graphics object that has been attached to a printer device, and whose clipping rectangle has been set to whole or part of the Component's bounding rectangle.

**Imports**

*java.awt.Graphics*

**Description**

This method prints the Component after calling validate. Note that this method is usually used to force a Container to repaint itself and all its child Components.

**Returns**

None.

**See Also**

The Graphics class; the validate method of the Component class

## repaint

**ClassName**

Component

**Purpose**

Requests an asynchronous repainting of the Component.

**Syntax**

public void repaint();

public void repaint(long lMillisecs);
public void repaint(int x, int y, int width, int height);
public void repaint(long lMillisecs, int x, int y, int width. int height);

**Parameters**

***long lMillisecs***

Maximum number of milliseconds to wait before the Component's update method is called.

***int x***

***int y***

***int width***

***int height***

These four parameters define a rectangle of area that should be repainted.

**Description**

You call this method at any time to force an asynchronous repainting of the Component. The Java system schedules a repainting of the Component to be completed by a different Thread at a later time. The second and fourth overloaded version of this method specify a maximum amount of time for the system to wait to schedule a repainting of the Component. The third and fourth overloaded versions allow you to specify a subset of the Component's bounding rectangle to repaint. Unless you use one of the overloaded versions of this method, which allows you to specify a maximum time limit, there is no guarantee on the amount of time before a repaint will be performed. If multiple repaint calls are made in quick succession, they will be combined into a single repainting operation. Repainting is achieved by an asynchronous call to update by the Java system.

**See Also**

The update and paint methods of the Component class

**Example**

See the example under the mouseDown method of the Component class.

## requestFocus

**ClassName**

Component

**Purpose**

If possible, gives the input focus to this Component.

**Syntax**

public void requestFocus();

**Parameters**

None.

**Imports**

None.

**Description**

Makes a request for the keyboard focus to be switched to this Component. This Component will be notified by a call to gotFocus when the keyboard focus has been switched. Note that disabled Components can not gain the keyboard focus.

**Returns**

None.

**See Also**

The nextFocus, gotFocus, and lostFocus methods of the Component class

**Example**

This example Container has two child Components, a text field, and a checkbox. When the checkbox becomes checked, then the text field is enabled and keyboard focus is given to it. When the checkbox becomes unchecked, then the text field becomes disabled and keyboard focus is given to the checkbox.

```
public class FocusContainerExample extends Panel {
    TextField _tf = new TextField();
    Checkbox _cb = new Checkbox("Enable text field");

    public FocusContainerExample() {
        add(_cb);
        add(_tf);
    }

    // An action Event is given to the parent Container
    // when the checkbox is checked or unchecked.
    action(Event evt, Object arg) {
        boolean b = ((Boolean)arg).booleanValue();
        if(!b) {
            _tf.disable();
            _cb.requestFocus();
        } else {
            _tf.enable();
            _tf.requestFocus();
        }
    }
}
```

## reshape

**ClassName**

Component

**Purpose**

Changes the origin and dimensions of the Component object in one method call.

**Syntax**

public void reshape(int x, int y, int width, int height);

**Parameters**

***int x***
***int y***
***int width***
***int height***

These four parameters describe a new bounding rectangle for the Component. The *x* and *y* coordinates are relative to the upper-left corner of the parent Container.

**Imports**

None.

**Description**

Modifies the bounding rectangle of the Component. The move and resize methods are actually wrappers around the reshape method. To detect when your (non-Frame) Component object is being resized or moved, re-implement the reshape method to set some detection flag before calling the base implementation, as demonstrated in the example below.

**Returns**

None.

**See Also**

The move and resize methods of the Component class

**Example**

This example Container re-implements reshape so that it can detect when it is being sized below a particular width and change its LayoutManager accordingly.

```
public class MyContainer extends Panel {
    ...
    static final int MIN_FLOWCENTER_WIDTH = 100; // any val

    public void reshape(int x, int y, int width, int height) {
        if(width < MIN_FLOW_WIDTH)
            setLayoutManager(new FlowLayout(FlowLayout.LEFT));
        else
            setLayoutManager(new FlowLayout(FlowLayout.CENTER));
        super.reshape(x, y, width, height);
    }

    ...
}
```

## resize

**ClassName**

Component

**Purpose**

Changes the dimensions of this Component.

**Syntax**

public void resize(int width, int height); public void resize(Dimension dim);

**Parameters**

***int width***

***int height***

The new width and height of the Component.

***Dimension dim***

The *width* and *height* members of this object describe the new width and height of the Component.

**Imports**

*java.awt.Dimension*

**Description**

Modifies the width and height of the bounding rectangle for this Component to be *width* pixels in width and *height* pixels in height. Note that the resize method is just a wrapper around the reshape method.

**See Also**

The Dimension class; the move and reshape methods of the Component class

**Example**

This example Component resizes itself to always be large enough to display a particular string.

```
public class MyComponent extends Canvas {
    ...

    String _str;

    public void setString(String str) {
        _str = str;
        FontMetrics fm = getFontMetrics(getFont());
        resize(fm.stringWidth(_str), size().height);
    }

    ...
}
```

## setBackground

**ClassName**

Component

**Purpose**

Sets the background color used to erase the Component when it is rendered.

**Syntax**

public void setBackground(Color c);

**Parameters**

***Color c***

The background color to use when rendering the Component in the future.

**Imports**

None.

**Description**

Sets the background Color to use when painting or printing this Component on a drawing surface. The update method uses a Component's background Color to erase the Component's bounding rectangle on the desktop before calling paint. If update is re-implemented so that it does not erase the Component, then the background Color is never used and might as well never be set.

**Returns**

None.

**See Also**

The Color class; the update method of the Component class

**Example**

See the example under the mouseEnter method of the Component class.

## setForeground

**ClassName**

Component

**Purpose**

Sets the foreground color used for rendering in the paint method.

**Syntax**

public void setForeground(Color c);

**Parameters**

***Color c***

The background color to use when rendering the Component in the future.

**Imports**

*java.awt.Color*

**Description**

Sets the foreground Color to use when painting or printing this Component on a drawing surface. update modifies the foreground Color used by the passed Graphics to be *c* before passing the Graphics on to paint. Therefore, Components that override the default implementation of update must set the foreground color explicitly in paint or update.

**Returns**

None.

**See Also**

The Color class; the getForeground, update, and paint methods in the Component class

**Example**

See the example under the mouseEnter method of the Component class.

## show

**ClassName**

Component

**Purpose**

Makes the Component either hidden or unhidden, according to the parameters passed.

**Syntax**

public void show(); public void show(boolean fShow);

**Parameters**

***boolean fShow***

True if the Component should be unhidden. False if it should be hidden.

**Imports**

None.

**Description**

Shows the Component. Hidden components are not drawn, nor do they take up space on the display surface. The second overloaded version allows you to hide or show the Component based on the value of *fShow*.

**Returns**

None.

**See Also**

The hide method of the Component class

**Example**

This example Component gets hidden whenever it is disabled.

```
public class MyComponent extends Canvas {
    ...

    public void disable() {
        show(false);
    }

    public void enable() {
        show(true);
    }
    ...
}
```

## size

**ClassName**
Component
**Purpose**
Gets the dimensions of this Component object.
**Syntax**
public Dimension size();
**Parameters**
None.
**Imports**
None.
**Description**
Gets the width and height of this Component.
**Returns**
A Dimension object whose *width* and *height* public member variables contain the Component width and height in pixels, respectively.
**See Also**
The bounds method of the Component class
**Example**
See the examples under the methods resize, getParent, and getForeground of the Component class.

## toString

**ClassName**
Component
**Purpose**
Creates a descriptive string detailing the internal state of the Component.
**Syntax**
public String toString();
**Parameters**
None.
**Imports**

None.

**Description**

Gets a String containing a textual description of this Component object. The resultant String is a concatenation of the object's class, and certain information about the Component's internal state such as whether or not it is enabled or hidden.

**Returns**

A textual description in a String object.

**See Also**

The toString method of the Object class

## update

**ClassName**

Component

**Purpose**

Called by the Java system whenever the Component should repaint itself.

**Syntax**

public void update(Graphics g);

**Parameters**

***Graphics g***

A Graphics object attached to the Component's display device, with a clipping rectangle equal to or a subset of the Component's bounding rectangle.

**Imports**

*java.awt.Graphics*

**Description**

Called automatically by the system when it is time to render the Component on a drawing surface. Calls to repaint cause an asynchronous call to update to be made by a separate Thread. The default implementation of update passes the Graphics object on to paint after erasing the entire drawing surface with the background color and selecting the foreground color into the Graphics object. Many Components, which require a lot of graphical updating, override update so that the entire drawing surface will not be erased. This prevents the Component from appearing to flicker with each graphical update.

**See Also**

The Graphics class; the paint method of the Component class

**Example**

See the example for the paint method of the Component class.

## validate

**ClassName**

Component

**Purpose**

To clear the invalid flag of this Component.

**Syntax**

public void validate();

**Parameters**

None.

**Imports**

None.

**Description**

Forces the Component to validate itself. When created, Components are marked as valid. Subsequent calls to the invalidate method mark the Component as invalid. Invalid Components validate themselves by calling the layout method before clearing their internal invalid flag. The default implementation of layout does nothing. The invalidate/validate methods are used mostly by Containers to force subcomponents to be laid out by a LayoutManager object.

**Returns**

None.

**See Also**

The invalidate and layout methods of the Component class; the layout method of the Container class

## The Component Project: A Hotspot Custom Component

The Component Project illustrates the construction of a simple custom Component class: the Hotspot class. The Hotspot class is a Component that has two Images associated with it: an ActiveImage and an InactiveImage. The behavior of a Hotspot object is to display the ActiveImage when the mouse cursor is moved on to the Hotspot. When the mouse is not over the Hotspot, the InactiveImage is displayed. This is a generic custom Component suitable for use in your own Java applications or applets.

This project demonstates several key concepts of Components and custom Components:

- Rendering a custom Component by overriding the Component.paint() method.
- Event handling by implementing Hotspot.mouseEnter() and Hotspot.mouseExit() to make the Hotspot react to user interaction. The Hotspot's Container—the HotsportApplet in this project—can also handle mouse click Events originally delivered to the Hotspot objects.
- Image preparation. The Hotspot implements the ImageObserver interface so that it can be notified of the progress of construction of the on-screen representation of the ActiveImage and the InactiveImage.

Figures 2-6 and 2-7 show the active and inactive images used for one of the Hotspot components in this project.

**Figure 2-6** Active image for the first Hotspot component of the Hotspot project

**Figure 2-7** Inactive image for the second Hotspot component of the Hotspot project

## Assembling the Project

**1.** Create a file called Hotspot.java using a text editor. This file holds the implementation of the Hotspot custom Component class. Begin by declaring the class and its member variables:

```
import java.awt.*;
import java.awt.image.*;

public class Hotspot extends Canvas implements ImageObserver {
    // The active and inactive images are stored in
    // member variables. A reference to the Image currently
    // being displayed is also kept as a member variable.
    Image _imgActive;
    Image _imgInactive;
    Image _imgCurrent;
```

**2.** The Hotspot constructor is passed a reference to the active and inactive Images. The Constructor stores those references in member variables and begins preparing the Images for rendering:

```
public Hotspot(Image imgActive, Image imgInactive) {
        _imgActive = imgActive;
        _imgInactive = imgInactive;
        _imgCurrent = _imgInactive;

        // prepareImage() starts the construction of an
        // on-screen representation of the image objects.
        prepareImage(_imgActive, this);
        prepareImage(_imgInactive, this);
    }
```

**3.** When the mouse moves over the Hotspot, the active image should be displayed. This is performed by the mouseEnter event handler, which is called automatically whenever a MOUSE_ENTER Event is delivered to the Component. Similarly, when the mouse is no longer over the Component, then the inactive image is displayed. This is performed by the mouseExit Event handler, which is called whenever a MOUSE_EXIT Event is delivered to the Component.

```
public boolean mouseEnter(Event evt, int x, int y) {
        _imgCurrent = _imgActive;
        repaint();
        return true; // the Event has been handled.
    }

public boolean mouseExit(Event evt, int x, int y) {
        _imgCurrent = _imgInactive;
        repaint();
        return true; // The eEvent has been handled.
    }
```

**4.** To reduce flicker, prevent default implementation of update from erasing the Component before calling paint.

```
public void update(Graphics g) {
```

```
        paint(g);
    }
```

**5.** Painting the Hotspot Component merely involves displaying the Image referred to by *_imgCurrent*.

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

**6.** Create a second file named HotspotApplet.java in the same directory. This file holds a sample Applet which displays two Hotspots. The following code initializes and displays two Hotspot Components on the surface of an Applet.

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

public class HotspotApplet extends Applet {
    Hotspot _hotspot1;
    Hotspot _hotspot2;

    // Applet initialization is the only operation that must
    // be implemented for this Applet class. The Active and
    // Inactive image URLs are passed as the "ActiveImageURL1",
    // "ActiveImageURL2", "InactiveImageURL1", and
    // "InactiveImageURL2" parameters. These four image URLs
    // are used to create two Hotspot components, which are
    // added as Components to the Applet (which acts as the
    // Container).
    //
    // This applet uses a FlowLayout object, which essentially
    // positions Components left-to-right in a line across
    // the Applet.
    public void init() {
        URL urlActive1;
        URL urlActive2;
        URL urlInactive1;
        URL urlInactive2;

        setLayout(new FlowLayout());
        resize(450, 200);
        setBackground( Color.black );

        try {
            urlActive1 = new URL(getDocumentBase(),
                    getParameter("ActiveImageURL1"));
            urlActive2 = new URL(getDocumentBase(),
                    getParameter("ActiveImageURL2"));
            urlInactive1 = new URL(getDocumentBase(),
                    getParameter("InactiveImageURL1"));
            urlInactive2 = new URL(getDocumentBase(),
                    getParameter("InactiveImageURL2"));
        } catch(Exception e) {
            System.out.println( "Image URLs are missing or " +
                "malformed." );
            return;
        }
```

```
        // Create the Image objects from these URLs.
        Image imgActive1 = getImage(urlActive1);
        Image imgActive2 = getImage(urlActive2);
        Image imgInactive1 = getImage(urlInactive1);
        Image imgInactive2 = getImage(urlInactive2);

        // Create the two Hotspot objects and add them
        // as Components of this Applet Container. Resize
        // the hotspots to the size the images
        // are expected to be.
        _hotspot1 = new Hotspot(imgActive1,
            imgInactive1);
        _hotspot2 = new Hotspot(imgActive2,
            imgInactive2);

        add( "1", _hotspot1 );
        add( "2", _hotspot2 );

        _hotspot1.resize(204, 140);
        _hotspot2.resize(204, 140);

        return;
    }

    // Re-implementation of the mouseDown() event handling
    // method allows us to react to the user clicking on
    // one of the Hotspot Components.
    public boolean mouseDown(Event evt, int x, int y) {
        if(evt.target == _hotspot1) {
            System.out.println(
                "The first hotspot was clicked!" );
            return true;
        }

        if(evt.target == _hotspot2) {
            System.out.println(
                "The second hotspot was clicked!" );
            return true;
        }

        return false; // Event was not handled.
    }
}
```

**7.** Create a file named Hotspot.html. This is an HTML file with an embedded HotspotApplet in it. Copy the following text to your Hotspot.html file:

```
<HTML>
<HEAD>
<TITLE>HotspotApplet Sample Project</TITLE>
</HEAD>

<HTML>
<HEAD>
<TITLE>HotspotApplet Sample Project</TITLE>
</HEAD>

<BODY>
```

```
<APPLET CODE="HotspotApplet.class" WIDTH=450 HEIGHT=200>
<PARAM NAME="ActiveImageURL1" VALUE="active1.gif">
<PARAM NAME="ActiveImageURL2" VALUE="active2.gif">
<PARAM NAME="InactiveImageURL1" VALUE="inactive1.gif">
<PARAM NAME="InactiveImageURL2" VALUE="inactive2.gif">
</APPLET>
</BODY>
</HTML>
```

**8.** Compile HotspotApplet.java using the JDK's *javac* compiler. From the directory where your .JAVA and .HTML files are located, run this command:

```
> javac HotspotApplet.java
```

**9.** Create four .GIF files to act as your Active and Inactive images. The images displayed above can be used. They are located on the Java API SuperBible CD under the directory \FOO\BLAH\WHATEVER. Copy these four files to the same directory your .JAVA and .HTML files are in. Make sure the names of these four files are "active1.gif", "active2.gif", "inactive1.gif" and "inactive2.gif", respectively.

**10.** Load the HotspotApplet into the JDK's AppletViewer. From the same directory where your project files are, type this command:

```
> appletviewer Hotspot.html
```

## How It Works

The Hotspot class maintains an internal reference to two Image objects. The *Hotspot._imgActive* member variable is a reference to an Image to display when the mouse cursor is over the Hotspot component. The *Hotspot._imgInactive* member variable is a reference to an Image to display when the mouse cursor is not over the Hotspot. The *Hotspot._imgCurrent* member variable is a reference to the Image to display: either the active or the inactive Image.

The Hotspot constructor requires references to the active and inactive Image as parameters. Part of object construction includes kick-starting construction of on-screen representation of both these Images using Component.prepareImage(). Always, before an Image can be drawn on a display device, the Image must be prepared by the Java system for rendering on the display device. The Hotspot object uses its inherited implementation of the ImageObserver interface, which is implemented by the Component class. The Component class' implementation of this interface causes the Component to transparently schedule a full repainting of the Hotspot component once the Image has been prepared.

The value of *_imgCurrent* changes whenever the mouse cursor either enters or exits the bounding rectangle of the Hotspot. Overriding implementations of the mouseEnter and mouseExit Event handling methods are used to detect the position of the mouse cursor. Through the Event delivery methods, the Java system calls a Component's mouseEnter() method exactly once, when the mouse is moved from outside the Hotspot's bounding rectangle to within it. When the mouse cursor is moved from inside to outside the

Hotspot's bounding rectangle, the mouseExit method is also called exactly once. Each Hotspot implementation of these methods performs three important tasks:

- Modifies the value of *_imgCurrent* to indicate either the active or inactive Image.
- Schedules a repainting of the Hotspot using the parameterless repaint() method.
- Returns true from the Event handling method, indicating that the Event has been handled and can be discarded.

Hotspot.paint() is implemented to paint the Hotspot's display surface with the Image indicated by *Hotspot._imgCurrent*. Before this can be done, the Hotspot must make sure the Image has been fully prepared for display on the device. checkImage() will return a logical ORing of the ImageObserver flags indicating which data for the Image has been successfully prepared for rendering. Hotspot checks for the ALLBITS flag, which indicates the Image has been fully prepared. Once this flag is detected, the Hotspot draws the Image.

Figure 2-8 is a screenshot of the the HotspotApplet running within the JDK's AppletViewer. Note that the mouse cursor is over the first of the two Hotspot Components, so the first Hotspot is displaying its Active image, while the second Hotspot is displaying its Inactive image.

**Figure 2-8** Screenshot of the HotspotApplet running within the JDK's AppletViewer

## mouseMove

**ClassName**

Component

**Purpose**

The Event handler for MOUSE_MOVE Events.

**Syntax**

public boolean mouseMove(Event evt, int x, int y);

**Parameters**

***Event evt***

The MOUSE_MOVE Event object that was passed to this Component's handleEvent method.

***int x***

***int y***

The x and y coordinates, relative to the Component's origin, where the mouse cursor was moved to.

**Description**

The mouseMove method is called by the default implementation of the handleEvent method of the Component class whenever a MOUSE_MOVE Event

is passed to that method. By handling MOUSE_MOVE Events in the mouseMove method, your Component can detect where the mouse cursor is currently positioned within the Component. The first MOUSE_MOVE Event is passed to your Component only after a MOUSE_ENTER Event (MOUSE_ENTER Events can be handled by the mouseEnter Event handler). The MOUSE_EXIT Event, handled by the mouseExit Component method, indicates that the mouse cursor has left your Component's bounding rectangle on the desktop, and that you will no longer receive MOUSE_MOVE Events. If the mouse is being moved with the mouse button held down, then your Component will receive MOUSE_DRAG Events, handled by the mouseDrag Event handler method.

**Returns**

As with all Event handlers, your implementation of this method should return true if the Event is completely handled by your code. A return value of false will cause the Event to be passed on to your Component's Container through the Event passing mechanism described earlier in this chapter.

**See Also**

See the mouseDrag, mouseEnter, and mouseExit methods of the Component class.

**Example**

This example demonstrates simple handling of MOUSE_MOUSE Events using a mouseMove method implementation. The method just writes the mouse position to System.out.

```
public class MyComponent extends Canvas {
    ...

    public boolean mouseMove(Event evt, int x, int y) {
        System.out.print("Mouse move detected.");
        System.out.println(" Current mouse position is: ("+
                x + ", " + y + ")");
    }

    ...
}
```

# *Part II*
# *Windows And Text Handlers*

# Chapter 3
# Toolkit, Window, Container, And Events

One of the challenges of designing Java was to make a programming system that was compatible across different operating systems. Making a system that is also compatible across the various windowing platforms complicates this problem quite a bit. One of Java's greatest strengths is the architecture of its windowing classes, which successfully achieve the goal of cross-platform compatibility between windowing systems.

Java has been ported into several windowed operating systems: Microsoft Windows 95 and Window NT, Solaris with X-Windows/Motif, and Macintosh System 7. While these systems are all based on similar concepts of how a windowing system should look and behave, each has different underlying architecture and implementation details. Java cuts through the confusion among the various systems, providing a consistent presentation for all of them.

No matter which windowing system a Java application is actually running on, the system is accessed through the same generic set of Java classes. These classes have been abstracted so that they will work on any of the supported windowing systems. This abstraction layer, which sits between Java applications and the windowed operating system, is called the Abstract Windows Toolkit, or AWT for short. Figure 3-1 shows how the AWT classes present a consistent set of API methods to Java applications and applets regardless of what operating system is actually running the application.

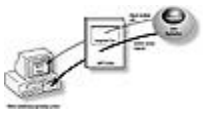

**Figure 3-1** The Abstract Windows Toolkit represents the windowed operating system to Java applications

The AWT is described in detail in this chapter. The window hierarchy, upon which is based both Event notification routing and on-screen positioning of Containers and Components, is detailed. This chapter also covers how Events are delivered to Components and Containers by the Toolkit, as well as how on-screen repainting operations are scheduled and carried out. The other services provided by the Toolkit are also discussed.

The project for this chapter, FontLab, is an example of a Java application that utilizes the system-wide services of the Toolkit to catalogue the Fonts available on a particular system. FontLab also demonstrates how Component z-ordering, or overlaying of sibling Components on top of one another, is done in Java.

## A Window in Java

No matter what operating system you are using, the basic unit of a windowed user interface is, of course, the window. What is a generic window? Some basic attributes are obvious. First, a window occupies a rectangular area of the desktop surface. Second, a window's rectangle can be moved and resized to change its position and appearance on the desktop. Third, a window can be embedded within a parent window. In fact, all top-level windows can be said to be children of the desktop, which is just another large window. In Java, these three attributes have been abstracted to the Component class, which is discussed in Chapter 2.

### Windows as Pseudo-I/O Devices

An application's window can be thought of as the device the application uses to gather input from the user, just like any other I/O device. One of the most important and basic services a windowed operating system provides is a way to gather information about a user's actions. That is, if the user clicks the mouse on a window, it is the job of the windowed operating system to provide that input to the window's application. Only after the application has received a report of the user's actions can it react. Figure 3-2 shows how, among other I/O devices, an application's window is used to gather user interactions.

**Figure 3-2** An application's window used to gather user interactions

In Java, user interactions are called events. An Event is an object that contains information about a single action. One of the most important tasks of the Java AWT is to deliver Events to your Java application or applet windows.

The Component class has one public method to receive Events: postEvent. Again, Chapter 2 is dedicated to disussing the Component class in detail, so the particulars of the postEvent method are discussed there. In that chapter is a description of what happens to the Events after they have been delivered to a Component object. The question is, How do Events get delivered to Components in the first place?

It is the job of the AWT to translate user interactions within the windowed operating system into deliverable Events within Java. To accomplish this, the AWT uses a proxy architecture. Each Component object in a Java application is mirrored by an object in the windowed operating system. This object is a native item in the windowed operating system, whether that be a window in MS Windows, or a widget in the X-Windows/Motif operating system. Within AWT, a ComponentPeer object acts as the go-between between the Java Component object and the native operating system object. When a user interaction is posted to the native operating system object, the AWT and the ComponentPeer translate that user action into an Event. That Event is then handed to the Component object through its postEvent method.

More specifically, the AWT manages a Thread dedicated to delivering Events to Components. Because of their source (the user) Events are always asynchronous in nature. Figure 3-3 is a screenshot of the System.out output of a very simple Java application, which intentionally throws an exception when it receives an Event. This application has a window that throws an ArrayIndexOutOfBounds Exception in its postEvent implementation. The output displayed in Figure 3-3 is a stack-trace of the uncaught Exception. Note that the name of the Thread in which the Exception was thrown is AWT-Callback-Win32. This illustrates the name of the AWT Thread responsible for delivering Events to windows. In this case, the underlying windowed operating system is Windows NT. Here is the code for the simple application, called Spike, which was used to generate the Exception.

**Figure 3-3** Stack-trace of the Spike program

```
import java.awt.*;

public class Spike {

    public static void main(String[] astrArgs) {
        SpikeFrame s = new SpikeFrame("Spike");
        s.show();
    }
}

class SpikeFrame extends Frame {

    public SpikeFrame(String strTitle) {
        super(strTitle);
    }

    public boolean postEvent(Event evt) {
        // Throw Array indexOutOfBounds Exception
        // by dereferencing a non-existent Component.
        return getCompoent(0).postEvent(evt);
    }
}
```

## Events

An Event object is designed to encapsulate any type of user interaction. Mouse events such as mouse movements and clicks, keyboard events like keypresses or keyreleases, and other interactions are each encodable in a single Event object.

An Event object exposes its member variables rather than forcing you to access the variable through member methods. Though the object-oriented programming rule-of-thumb is that member variables shouldn't be available for other objects to modify directly (i.e., without at least using member methods to modify the values), Event objects are simple enough to forgo this kind of strict access control. In addition, it cuts down on code size to say *myEvent.id* rather than *myEvent.getID()*.

The *id* member variable indicates which type of Event has occurred. Here are the possible values for *Event.id* which are specific to Window objects, that is, the values that indicate events created by Java.

| Event Type | Description |
|---|---|
| WINDOW_DESTROY | The window has received a command to destroy itself. Only |

| | |
|---|---|
| | Window class objects will receive this type of Event. Call Window.dispose() to properly destroy the Window. |
| WINDOW_EXPOSE | The window has become exposed. |
| WINDOW_ICONIFY | The window has become iconified. |
| WINDOW_DEICONIFY | The iconic window has become de-iconified. |
| WINDOW_MOVED | The window has been moved (and possibly resized). |

---

There is nothing stopping an application from creating its own custom Event types and passing them to Component windows. The Component class description in Chapter 2 describes the keyboard and mouse Events which may be sent to a Component object. In Chapters 4 and 5 the Event IDs reserved for Scrollbar and List objects are discussed in detail.

## Window Repainting in AWT

As suggested earlier, a window can be thought of as an Input/Output device. AWT's architecture creates an efficient system for delivering Events to windows. Events are the Input end of a window. A window is also an output device. The output is what the window displays on the surface of its rectangle.

All of the windowing operating systems handle painting windows in a similar manner. The windowing system determines when a window must repaint itself based on window management. For example, if two windows are overlapping and the window on top is removed, then the windowing system flags that the remaining window must be repainted as soon as possible. (Actually, just the part of the remaining window which was uncovered is flagged for repainting.)

The repaint operation is an asynchronous operation for a couple of reasons. First, the system usually determines a window must be repainted as a side effect of some operation the system is trying to perform. In this case, the repainting operation must be scheduled for some time in the future. Second, repainting can be a time-intensive procedure. The system attempts to store as many redundant repainting operations for the same window as possible. When the system thinks there is enough idle time to repaint the window, then it will explicitly ask the window to repaint itself.

In the AWT, a window is requested to repaint its surface via Component.update(). Chapter 2 explained what happens within the Component class after the system calls update(), but what causes update() to be called in the first place? A similar callback mechanism to the Event delivery procedure described earlier is used. When the underlying windowing system issues a repaint command to the native window object associated with a Java Component, the AWT translates this to a call to the Component's update() method.

Figure 3-4 shows the Spike2 application. Like the previous Spike application, Spike2 throws an ArrayIndexOutOrBoundsException in the update() method of its main window. Figure 3-4 also includes a stack-trace at the time the exception is thrown. Notice the same Thread, AWT-Callback-Win32, controls the repainting operation as controlled by the Event delivery operation in Spike. Here is the code for the Spike2 application.

**Figure 3-4** The Spike2 application

```
import java.awt.*;

public class Spike2 {

    public static void main(String[] astrArgs) {
        Spike2Frame s = new Spike2Frame("Spike2");
        s.show();
    }
}

class Spike2Frame extends Frame {

    public Spike2Frame(String strTitle) {
        super(strTitle);
    }

    public void update(Graphics g) {
        // Throw an ArrayIndexOutOfBounds
        // Exception by dereferencing a non-existent
        // Component.
        getComponent(0).update(g);
    }
}
```

Of course, rather than throwing Exceptions during repainting or Event handling methods, your code should handle each call as quickly as possible. Stalling or suspending the callback Thread will adversely affect your application in unforeseen ways.

## Components, Containers, and Top-Level Windows

The Component class defines a child window. That means that a simple Component window must exist as a child to a parent window. The Component class is written so that a Component does not have its native windowing system peer created unless the Component has been added to a parent window. This is reflected in the Java API by the fact that you can't display a simple Component, such as a Canvas or Scrollbar, on the desktop without a parent window.

### Containers

A special subclass of Component is the Container class. A Container window is a type of Component that can be a parent to other Components, including other Containers. To add a Component as the child of a Container, you use the Container class method add(), shown here:

```
Container cont;

// instantiate cont to be a container, such
// as a Frame or a Panel.

Canvas c = new Canvas();

// c does not have a native windowing system peer yet,
// because it does not have a parent window.

cont.add(c);
// c's peer gets created automatically as a result of
// the call to add.
```

A Container contains zero or more Components. These Components are called siblings, since they have the same parent window. One important thing to remember is that sibling windows "clip" each other. That is, if you had two overlapping sibling windows, one of the windows appears on top of the other.

The term denoting the relative precedence of sibling Components is z-order. A Component with a higher z-order will appear on top of its overlapping siblings. The z-order of sibling Components is determined by the order in which they were added to the parent Container. The last Component added to a Container has the lowest z-order. Any Component added before another Component will appear on top if the two overlap within the parent Container.

Note that all Containers have a LayoutManager which arranges the child Components within the Container. The Java API includes several types of LayoutManagers to arrange child Components by different methods. For example, a FlowLayout object will arrange a Container's child Components side-by-side, left-to-right, top-to-bottom. Chapter 4 discusses the various LayoutManager classes.

Because the LayoutManager classes, included with the Java API, ensure that sibling Components never overlap within their parent Containers, our discussion of z-ordering is academic as long as you use only those LayoutManagers in your Containers. However, in Containers that do have overlapping child Components (as would be the case if you implemented your own LayoutManager to cause siblings to overlap, as is done in the chapter's project) z-ordering can be important.

You can also remove a Component from its Container. The remove method takes, as a reference, the Component you want removed as a child for the Container. When the Component is removed from its parent Container, the Component's native windowing system peer is automatically destroyed. Again, a Component can not have a native peer object unless the Component has a parent.

The following AddButton application demonstrates the use of Container.add() and Container.remove().The AddButton application includes a “+” and “-” push button. Press the “+” button to create a new button Component. Press the “-” button to destroy the oldest button Component. The other buttons do nothing. Figure 3-5 shows the AddButton application.


**Figure 3-5** The AddButton application

```
import java.awt.*;

public class AddButton {

    public static void main(String[] astrArgs) {
        AddButtonFrame f = new AddButtonFrame("Add Button
Application");
        f.show();
    }
}

class AddButtonFrame extends Frame {
    Button _buttonAdd;
    Button _buttonRemove;
    int    _nButtonNum = 1;

    public AddButtonFrame(String strTitle) {
        super(strTitle);
        setLayout(new FlowLayout());

        // Declare add and remove buttons. Note that even
        // though the Button objects have been created, the
        // Button peers in the native windowing system have
        // not, since the Buttons have not been added to a
        // Container yet.

        _buttonAdd = new Button("+");
        _buttonRemove = new Button("-");

        // Add the "+" and "-" buttons. Once added to
        // this Frame, the buttons' peers in the native
        // windowing system are created.
        add(_buttonRemove);
        add(_buttonAdd);
    }

    public boolean action(Event evt, Object what) {
        // Make sure 'what' is a String.
        if(!(what instanceof String))
            return false;

        // If the add button was pressed…
        if("+".equals((String)what)) {
            // NOTE: addition of string and int converts int.
```

```
            add(new Button(""+_nButtonNum++),
                    countComponents()-1);
            validate();
        }

        // If the remove button was pressed…
        if("-".equals((String)what)) {
            remove(getComponent(1));
            validate();
        }

        return false;
    }
}
```

## Windows

As stated above, a Container is a special type of Component that can be a parent to zero or more child Components, including other Containers. A Container is still a Component, however, and, as such, the Container must also have a parent window.

The Window class is a subclass of Container that defines a top-level window. Top-level windows do not have to have parent windows. The native windowing system peer for a Window object is a pop-up window on the desktop. Therefore, when creating your user interface in Java, all Component objects must eventaully be descended from a Window object in the window hierarchy.

A Frame is a special type of Window. A Frame is a native windowing system top-level frame window, which has a titlebar, an optional menubar, and a resizable border. Note that in the sample applications in this chapter, the interface is always controlled by an object derived from Frame. That Frame is the application's main window.

## Peers and the Toolkit

How do the native windowing system objects, the peers, get created? Where do they come from? Earlier, we said that Component objects (execpt Window and Frame objects) do not have a peer created for them in the native windowing system until the Component is added as a child of a Container. Another way of putting it is: The Component does not have a peer created on its behalf until it is added to a Container *that has a peer*.

A Component's own addNotify method is called to create the Component's peer. Component.addNotify() is called by the parent Container. This call can happen either in Container.add(), as soon as the Component is added to the Container, or in Container.addNotify(). In the case of addNotify(), when a Container's peer is created, the Container also tells its child Components to create their peers.

Simple Component objects, such as a Canvas, create their own peers in overriden implementations of addNotify(). The Java code for Canvas.addNotify() looks something like this:

```
CanvasPeer _myPeer = null;

public void addNotify() {
    myPeer = Toolkit.getDefaultToolkit().createCanvasPeer();
}
```

Toolkit is the java.awt.Toolkit class.The Toolkit class is the class that represents the capabilities of the underlying windowing system to Java objects. For example, the createCanvasPeer method used above, uses native function calls to the underlying windowing system to create a CanvasPeer object. The CanvasPeer is a representative of a native windowing object in Java.

Within the Toolkit class is a create method for each of the Component types in the Java API. There is a Toolkit.createCanvasPeer, Toolkit.createButtonPeer, Toolkit.createScrollbarPeer, and so on, defined in the Toolkit class. The actual peer classes are discussed in detail in Chapter 9. The point being made here is that each Component class object uses the Toolkit to create its peer, and the creation of the peer occurs within the overriding implementation of addNotify. The Frame class calls its own addNotify method within Frame.show. That is, as soon as the Frame is supposed to be shown on the screen, its peer is created.

## The Toolkit

The Toolkit represents the windowing system within Java code. Most of the Toolkit's public methods are dedicated to the creation of peer components. The native windowing system provides additional functionalities beyond simply creating and managing windows. The Toolkit also exposes some of these additional functionalitites. The Toolkit class currently has methods to provide services in three additional areas: desktop metrics, available font information, and image downloading/preparation.

The size and composition of the desktop surface can be of great importance to some applications. Through the Toolkit's public methods, you can find out more about the desktop. The following table lists the Toolkit's desktop metrics methods and provides a description of each.

| Method | Description |
|---|---|
| getScreenSize | Returns a Dimension object whose width and height is equal to the width and height of the desktop, in pixels. |
| getScreenResolution | Returns resolution of the desktop, in pixels-per-inch. |
| getColorModel | Returns the ColorModel of the desktop. If the system uses a 256-color display, then this would be an IndexedColorModel, |

which would give you read access to the system palette.

---

It is through the Toolkit that an application or applet can enumerate the fonts available on the system. Toolkit.getFontList() returns an array of Strings. Each element of the array is a typeface name for a font available on the system. To get the FontMetrics for a Font when it is used on the desktop screen, you can use getFontMetrics, passing in the Font that is to be measured. Note that FontMetrics are also available through Graphics.getFontMetrics.

The image methods included in the Toolkit, allow an application to download and display images. To create an Image object from a graphics-format file, use Toolkit.getImage(). Two overloaded versions of this method are provided. The first version takes a URL pointing to the network location of the graphics format file. The second version takes a file path name and loads the image from a file on the local file system.

The Image object returned from getImage represents the graphics format file to Java. Before the Image can be copied to a display surface, the Image must be fully "prepared," or constructed in memory. The Toolkit's prepareImage method is used to kick-start the Image construction process. prepareImage takes, as a parameter, an ImageObserver. The ImageObserver will be notified as to the progress of the Image construction process. Any errors in the graphics file will also be reported to the ImageObserver. After an Image has been fully prepared once, it can be drawn on any display surface any number of times. checkImage is used by objects, other than the ImageObserver, to get information on the progress of the Image construction process.

Chapter 1 on Applets and Graphics, discusses downloading and preparing images because image and audio data have such an important application over the Internet.

## Toolkit, Window, Container, and Event Summaries

Table 3-1 lists the classes summarized in this chapter and a short description of each. Table 3-2 lists the methods of the classes from Table 3-1 and provides a short description of each method.

**Table 3-1** Class summaries

| Class | Description |
|---|---|
| Container | A Container is a special type of Component object which can be a parent to zero or more Components, including other Containers. The add() and remove() methods are used to manage the list of child components. countComponent() and getComponent() provide read-access to a Container's list of Components. |

| | |
|---|---|
| Event | User-interaction events are delivered by the AWT to Components in Event objects. |
| Window | A top-level pop-up window. Does not have a title bar or menu bar. This is the base class for all top-level windows. |
| Toolkit | Abstracts the functionalities of the native windowing system. The majority of methods are used to create peer native window objects or “peers” for Java Component objects. |

**Table 3-2** Summary of methods

| Class | Method | Description |
|---|---|---|
| Container | countComponents | Gets the number of child Components for a Container. |
| | getComponent | Gets a reference to a specific child of a Container. |
| | getComponents | Gets an array that enumerates all the child Components of a Container. |
| | insets | Gets the Insets objects that describe the border spacing around a Container. |
| | add | Adds a Component as a child of a Container. |
| | remove | Removes a child Component from a Container. |
| | removeAll | Removes all child Components from a Container. |
| | getLayout | Gets a Container’s LayoutManager. |
| | setLayout | Sets a Container’s LayoutManager. |
| | layout | Arranges a Container’s child Components. |
| | validate | Called when a Container should validate its sizing and positioning, and that of its child Components |
| | preferredSize | Gets the preferred size of the Container’s bounding rectangle. |
| | minimumSize | Gets the minimum size of the Container’s bounding rectangle. |
| | paintComponents | Performs a synchronous repainting of each of a Container’s child Components. |

| | | |
|---|---|---|
| | deliverEvent | Finds an Event handler to handle a specific Event. |
| | locate | Finds the child Component whose bounding rectangle includes a specific point. |
| Event | translate | Modifies the Event's *x* and *y* member variables. |
| | shiftDown | Indicates whether the SHIFT button was held down during the keyboard or mouse Event. |
| | controlDown | Indicates whether the CTRL button was held down during the keyboard or mouse Event. |
| | metaDown | Indicates whether the META button was held down during the keyboard or mouse Event. |
| Window | toBack | Sends the Window to the bottom of the desktop z-order. |
| | toFront | Brings the Window to the front of the desktop z-order. |
| | dispose | Destroys the Window's native windowing system peer. |
| | getWarningString | Gets the warning string displayed by Frame windows created by Applets. |
| Toolkit | createButton | Creates a ButtonPeer. The ButtonPeer knows how to translate button presses in the native windowing system to action Events. |
| | createTextField | Creates a TextFieldPeer. The TextFieldPeer knows how to manage a text field, select its contents and edit them, etc. |
| | createLabel | Creates a LabelPeer, which can get and set the text and alignment of a label. |
| | createList | Creates a ListPeer, which knows how to get and set the contents of a list and work with the selection in the native windowing system. |
| | createCheckbox | Creates a CheckboxPeer, which knows how to get and set the state of a native window system checkbox control. |
| | createScrollbar | Creates a ScrollbarPeer, which knows how to get and set the min, max and |

| | |
|---|---|
| | value of a native window system scrolllbar. The peer also translates scrollbar actions into Java Events for a Scrollbar object. |
| createTextArea | Creates a TextAreaPeer, which knows how to get and set the contents of a multiline text area in the native window system. |
| createChoice | Creates a ChoicePeer, which knows how to modify the contrents of a native window system choice box and which translates selection messages into Event for a Choice Component. |
| createFrame | Creates a FramePeer, which knows how to modify the titlebar of a native window system frame and can translate native window system window actions into Java Window events. |
| createCanvas | Creates a CanvasPeer, which can detect and translate mouse and keyboard user interactions. |
| createPanel | Creates a PanelPeer, which is much like a Canvas, except that it is also a Container. The PanelPeer knows how to add or remove Components to the Panel in the native window system. |
| createWindow | Creates a WindowPeer, which can create a simple top-level window and knows how to translate window messages from the native window system into Events for delivery in Java. |
| createDialog | Creates a DialogPeer, which can create a modal dialog in the native window system. |
| createMenuBar | Creates a MenuBarPeer, which knows how to modify the contents of menubars in the native window system. |
| createMenu | Creates a MenuPeer, which knows how to add and remove menu items from a native window system menu. |
| createMenuItem | Creates a MenuItemPeer, which knows how to check and modify the state of a MenuItem in the native window system. |

| | |
|---|---|
| createFileDialog | Creates a FileDialogPeer, which knows how to create a file dialog in the native window system and modify the contents of the variable file selection fields. |
| createCheckboxMenuItem | Creates a CheckboxMenuItemPeer, which knows how to create a menu item in the native windowing system and knows how to modify its state as it appears to be checked. |

## Container

**Purpose**

A Container is a Component which can contain other Components, including other Containers.

**Syntax**

public abstract class Container extends Component

**Description**

A Container is a Component that can contain other Components, including other Containers. A Container, being a Component, must be contained by another Container in order to be displayed. The Container class is abstract. The simplest type of realizable Container is a Panel. Figure 3-6 shows the class hierarchy of the Container class.

**PackageName**

*java.awt*

**Imports**

*java.io.PrintStream, java.awt.peer.ContainerPeer*

**Constructors**

None.

**Parameters**

None.

**Figure 3-6** The class hierarchy of the Container class

## countComponents

**ClassName**

Container

**Purpose**

Gets the number of child Components for this Container.

**Syntax**

public int countComponents();

**Parameters**

None.

**Description**

Returns the number of child Components in this Container.

**Imports**

None.

**Returns**

The number of Components that have been added to this Container using add.

**See Also**

The getComponent and getComponents methods of the Container class

**Example**

This example Container subclass demonstrates the use of both countComponents and getComponent to run through the list of a Container's child Components. The only method implemented by this class, getChildrenBounds returns a bounding rectangle of all the child Components.

```
class ContainerEx extends Container {

    public Rectangle getChildrenBounds() {
        Rectacngle rectRet = null;
        for( int ii=0 ; ii<countComponents() ; ii++ ) {
            Component c = getComponent(ii);
            Rectangle b = c.bounds();

            if(null == rectRet)
                rectRet = new Rectangle(b.x, b.y, b.width, b.height);

            rectRet.add(b);
        }

        return rectRet;
    }
}
```

## getComponent

**ClassName**

Container

**Purpose**

Gets a reference to a specific child Component of this Container.

**Syntax**

public Component getComponent(int index)

**Parameters**

***index***

Zero-based index of the Component to get. This must be between 0 and *(countComponents()-1)*.

**Description**

Gets a reference to one of the child Components of this comtainer. An ArrayIndexOutOfBoundException may be thrown if the *index* parameter is not valid.

**Imports**

*java.awt.Component*

**Returns**

A reference to the *index*-th Component child of the Container will be returned. Note that if the index parameter is less than 0 or greater than *(countComponents()-1)*, then an ArrayIndexOutOfBoundsException will be thrown.

**See Also**

The Component class; the countComponents and getComponents methods of the Container class

**Example**

See the example under countComponents.

## getComponents

**ClassName**

Container

**Purpose**

Gets an array of references to all child Components of this Container.

**Syntax**

public Component[] getComponents();

**Parameters**

None.

**Description**

Gets an array of Component, with one element for each of the child Components of the Container. The order of the Components in the array is the order the Components were added to this Container.

**Imports**

*java.awt.Component*

**Returns**

An array of Component objects. The length of this array will be the same as the return value from countComponents. Each child Component will appear in the array. Null may be returned if the Container has no children.

**See Also**

The Component class; the countComponents and getComponent methods of the Container class

**Example**

This is an alternative implementation to the getChildrenBounds function given in the example for the countComponents method.

```
class ContainerEx extends Container {

    public Rectangle getChildrenBounds() {
        Rectacngle rectRet = null;
```

```
        for( int ii=0 ; ii<countComponents() ; ii++ ) {
            Component c = getComponent(ii);
            Rectangle b = c.bounds();

            if(null == rectRet)
                rectRet = new Rectangle(b.x, b.y, b.width, b.height);

            rectRet.add(b);
        }

        return rectRet;
    }
}
```

## insets

**ClassName**

Container

**Purpose**

Specifies inset spacing between children and the Container's edge.

**Syntax**

public Insets insets();

**Parameters**

None.

**Description**

A Container can define an Insets object, which defines the border of padding within the Container. The Insets are used by the LayoutManager in such a way that no child Components will be placed within the Insets border.

**Imports**

*java.awt.Insets*

**Returns**

An Insets object is returned that describes the border for a LayoutManager to leave around the Container.

**See Also**

The LayoutManager class; the Insets class

**Example**

The default implementation of Insets simply delegates the call to the Container's peer. If the peer does not define an Insets, then Insets of 0 are returned. This implementation of insets and setInsets allows you to define your Container's Insets within Java code.

```
public ContainerEx extends Container {
    Insets _insets = new Insets(0, 0, 0, 0);

    public void setInsets(int left, int top, int right, int bottom) {
        _insets = new Insets(left, top, right, bottom);
    }

    public Insets insets() {
        return _insets;
    }
}
```

## add

**ClassName**

Container

**Purpose**

Adds a Component as a child of this Container.

**Syntax**

public void add(Component c);public void add(Component c, int index);

**Parameters**

***Component c***

The Component to add as a child of this Container.

***int index***

Index to store the Component in the Container's internal list of Components.

**Description**

Adds a Component as a child of this Container. The index of the Component, within the Container's list of Components, is set by the second parameter, *index*, of the second overloaded versions. No matter which version you use, the child Component has the lower z-order compared to its siblings. If the Component is currently a child of another Container, it will automatically be removed from the other Container before being added to this one. If the Component is a parent or ancestor of this Container, then an IllegalArgumentException will be thrown.

**Imports**

*java.awt.Component*.

**Returns**

None.

**See Also**

The Component class; the remove method of the Container class

**Example**

This example builds a simple toolbar of four buttons.

```
.// A Panel is a type of Container.
Panel p = new Panel();
p.setLayout(new FlowLayout());

p.add(new Button("Back"));
p.add(new Button("Forward"));
p.add(new Button("Home"));
p.add(new Button("Return"));
```

## remove

**ClassName**

Container

**Purpose**

Removes a Component as a child of this Container.

**Syntax**

public void remove(Component c);

**Parameters**

***Component c***

A Component which is a child of this Container.

**Description**

Removes a Component as a child of this Container. If the Component is not a child of this Container, then the call is ignored. The Component is added as a child of this Container using add.

**Imports**

*java.awt.Component*

**Returns**

None.

**See Also**

The Component class; the add method of the Container class

**Example**

This example removes all child Components and then adds them back in reverse order.

```
public class MyCont extends Panel {
    …

    public void ReverseChildren() {
        Component[] children = getComponents();
        removeAll();
        for( int ii=children.length ; ii>=0 ; ii-- )
            add(children(ii);
    }

    …
}
```

## removeAll

**ClassName**

Container

**Purpose**

Removes all child Components from this Container.

**Syntax**

public void removeAll();

**Parameters**

None.

**Description**

Removes all the child Components from this Container. Internally, this method is implemented by running through the list of Components and making repeated calls to remove.

**Imports**

None.

**Returns**

None.

**See Also**

The remove method of the Container class

**Example**

See the example under the remove method of the Container class.

## getLayout

**ClassName**

Container

**Purpose**

Gets the LayoutManager for this Container.

**Syntax**

public LayoutManager getLayout();

**Parameters**

None.

**Description**

Gets the LayoutManager for this Container. The LayoutManager is responsible for arranging the child Components within the Container's display rectangle. See Chapter 4 for a description of the LayoutManager and its relationship with the Container.

**Returns**

A reference to the Container's LayoutManager. Null if the Container has no LayoutManager.

**See Also**

The LayoutManager interface; the setLayout method of the Container class

**Example**

This example uses getLayout to display a Container's LayoutManager.

```
public class displayLayout(Container cont) {
    if(null != cont.getLayout())
        System.out.println(cont.getLayout());
}
```

## setLayout

**ClassName**

Container

**Purpose**

Sets the LayoutManager for this Container to use.

**Syntax**

public void setLayout(LayoutManager lm);

**Parameters**

***LayoutManager lm***

A LayoutManager to arrange the Components of this Container.

**Description**

Sets the LayoutManager which will arrange the child Components of this Container within the Container's bounding rectangle. See Chapter 4 for a description of the LayoutManager and its relationship with the Container.

**Imports**

*java.awt.LayoutManager*

**Returns**

None.

**See Also**

The LayoutManager interface; the getLayout method of the Container class

**Example**

This example Applet sets its own LayoutManager to a BorderLayout object.

```
public class MyApplet extends Applet {
    public MyApplet() {
        setLayout(new BorderLayout());
    }

    public void init() {
        add("Center", new Button("Go!"));
    }
}
```

## layout

**ClassName**

Container

**Purpose**

Called to allow the Container to arrange its child Components.

**Syntax**

public void layout();

**Parameters**

None.

**Description**

The default implementation of this method forces the LayoutManager to recalculate the placement of child Components within this Container's rectangle using the LayoutManager's layoutContainer method. The default implementation of the validate method of the Component class calls Component.layout. The Container class overrides layout with a custom implementation. See Chapter 4 for a description of the LayoutManager and its relationship with the Container.

**Imports**

None.

**Returns**

None.

**See Also**

The LayoutManager interface; the validate() method of the Component class

**Example**

In this example, a custom Container does not rely on a LayoutManager but instead arranges its child Components in an overridden layout implementation.

```
public class MyCont extends Panel {
    …

    public void layout() {
```

```
        // Layout children vertically, allowing each to be
        // its preferred height, but only as wide as this
        // Container.
        Component[] children = getComponents();
        Dimension dimThis = size();
        int y = 0;
        for( int ii=0 ; ii<children.length ; ii++ ) {
            Dimension dimChild =
                children[ii].preferredSize();
            children[ii].reshape(0, y, dimThis.width,
                    dimChild.height);
            y+=dimChild.height;
        }
    }

    …
}
```

## validate

**ClassName**

Container

**Purpose**

Called when the Container should validate is size and positioning and that of its child Components.

**Syntax**

public void validate();

**Parameters**

None.

**Description**

This overridden version of Component.validate does everything the Component version of this method does, plus it will validate all the child Components of this Container. A Container is invalidated by an explicit call to Component.invalidate. A Container is also invalidated whenever a Component is added to it or removed from it.

**Imports**

None.

**Returns**

None.

**Example**

See the AddButton example given earlier in this chapter.

## preferredSize

**ClassName**

Container

**Purpose**

Calculates the preferred size of this Container’s bounding rectangle using the Container’s LayoutManager.

**Syntax**

public Dimension preferredSize();

**Parameter**

None.

**Description**

This overridden implementation of Component.preferredSize calculates the Container's preferred size by asking the LayoutManager to calculate the preferred size in a call to LayoutManager.preferredLayoutSize. See Chapter 4 for a description of the LayoutManager and its relationship with the Container.

**Imports**

None.

**Returns**

A Dimension object whose *width* and *height* member variables hold the preferred size of this Container.

**See Also**

The preferredSize method of the Component class; the minimumSize method of the Container class

**Example**

See the example of the preferredSize method of the Component class (in Chapter 2).

## minimumSize

**ClassName**

Container

**Purpose**

Calculates the minimum acceptable size of this Container's bounding rectangle using the Container's LayoutManager.

**Syntax**

public Dimension minimumSize();

**Parameters**

None.

**Description**

This overridden implementation of Component.minimumSize calculates the Container's minimum size by asking the LayoutManager to calculate the minimum size in a call to LayoutManager.minimumLayoutSize. See Chapter 4 for a description of the LayoutManager and its relationship with the Container.

**Imports**

None.

**Returns**

A Dimension object whose *width* and *height* member variables hold the preferred size of this Container.

**See Also**

The minimumSize method of the Component class; the preferredSize method of the Container class

**Example**

See the example for the minimumSize method of the Component class (in Chapter 2).

## paintComponents

**ClassName**

Container

**Purpose**

Synchronously paints all children Components.

**Syntax**

public void paintComponents(Graphics g)

**Parameters**

***Graphics g***

The Graphics, associated with the display surface, to paint the Components on.

**Description**

Forces an immediate (synchronous) repainting of all the child Components. A synchronous repainting of each of the child Components is achieved by creating multiple clipped versions of the passed Graphics object (using Graphics.create), and passing the clipped versions to paint for each of the child Components to render itself.

**Imports**

*java.awt.Graphics.*

**Returns**

None.

**See Also**

The paint method of the Component class; the create method of the Graphics class

**Example**

In this example, Container's paint method is implemented by a simple call to paintComponents. This repaints the Container's child Components using the Graphics object passed to the Container's paint method.

```
public class MyContainer extends Panel {
…

    public void paint(Graphics g) {
        paintComponents(g);
    }

…
}
```

## deliverEvent

**ClassName**

Container

**Purpose**

Delivers an Event to a Container or one of its child Compnents.

**Syntax**

public void deliverEvent(Event evt)

**Parameters**

***Event evt***

The Event to deliver to this Container.

**Description**

This overridden version of Component.deliverEvent first passes the Event to the child Component indicated by the *x* and *y* member variables of the Event object. If the child Component does not handle the event, or the *x* and *y* Event member variables do not indicate a point within any of this Container's children, then the Event is posted to this Container through Container.postEvent.

**Imports**

*java.awt.Event.*

**Returns**

None.

**See Also**

The deliverEvent method of the Component class

**Example**

In this example, mouse events are successfully delivered to "virtual" (peerless) child Components of a Container using the Container's deliver Event method. The deliverEvent method of the Container class first attempts to post the Event to an appropriate child Component before letting the Container handle the Event.

```
//First, here's our peerless Component class
class PeerlessCanvas extends Canvas {
    public PeerlessCanvas() {}

    public void addNotify() {
        return;
    }

    public boolean mouseDown(Event evt, int x, int y) {
        // This code *will* be reached, because the
        // parent Container's deliverEvent will find
        // the correct child Component.
        …
        return true;
    }
}
// The Container class to make sure MOUSE_DOWN Events are
// posted to the correct, peerless, child Component.
public class PeerlessContainer extends Panel {
    public PeerlessContainer() {
        setLayout(new BorderLayout());
        add("North", new PeerlessCanvas());
        add("South", new PeerlessCanvas());
        add("East", now PeerlessCanvas());
        add("West", new PeerlessCanvas());
        add("Center", new PeerlessCanvas());
    }

    // mouseDown Event handler calls deliverEvent
    // to ensure mouse event is posted to correct
```

```
    // child Component, even if it's peerless.
    public boolean mouseDown(Event evt, int x, int y) {
        deliverEvent(evt);
    }
}
```

## locate

**ClassName**

Container

**Purpose**

Gets the child located at a particular point.

**Syntax**

Public Component locate(int x, int y);

**Parameters**

***int x, y***

Indicates a point relative to the Container's point of origin.

**Description**

Finds the Component which occupies the point, passed in the *x* and *y* parameters, to this method. The *x* and *y* parameters are expressed relative to this Container's origin.

**Imports**

*java.awt.Component*

**Returns**

The Component with lowest index in the Container's internal list of Components and which occupies a rectangle that the point falls into, is returned.

**Example**

This example lists the source code for the Container class' deliverEvent method. The deliverEvent method uses the locate method to find the correct child Component to deliver an Event to.

```
public void deliverEvent(Event evt) {
    Component c = locate(evt.x, evt.y);

    if ((c != null) && (c != this)) {
        evt.translate(-c.x, -c.y);
        c.deliverEvent(evt);
    } else {
        postEvent(evt);
    }
}
```

## Event

**Purpose**

Represents an asynchronous event which occurred in the system.

**Syntax**

public class Event

**Description**

Represents an asynchronous event which occurred in the system. For example, user-generated events like mouse moves or keyboard actions. The Event class is not extended by any class in the Java API, but rather the member variables of the Event class are sufficient for encoding any definable event. Figure 3-7 shows the class hierarchy of the Event class.

**PackageName**

*java.awt*

**Imports**

*java.io.**

**Constructors**

public Event(Object target, long when, int id, int x, int y, int key, int modifiers, Object arg);public Event(Object target, long when, int id, int x, int y, int key, int modifiers);public Event(Object target, int id, Object arg);

**Parameters**

The following table lists all of the Event class public member variables and a short description of each.

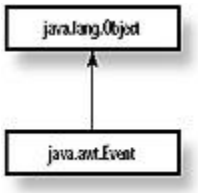


**Figure 3-7** The class hierarchy of the Event class

| Member Variable | Description |
|---|---|
| Object target | The Object to which the Event was originally passed. |
| long when | Time stamp of when the event occurred. |
| id | Identifies the type of the Event: mouse movement, keyboard action, etc. The following table lists all of the different Event types. |
| int x, y | A point where the Event occurred. These two variables are ususally only valid for mouse Events. |
| int key | The key pressed if this is a keyboard Event. |
| modifiers | Keyboard Event modifiers, such as whether or not the CTRL key was being held down, whether or not the SHIFT key was being held down, etc. |
| clickCount | For multiclicks (e.g., a double-click) this member indicates how many clicks took place. |
| Object arg | An arbitrary argument, which is different for each type of Event. |
| Event evt | The next Event. Used when storing Events in a linked list. |

The *id* member variable indicates what type of Event is being represented. Many different Event class constants have been defined to indicate Event types. The following table list the possible values for the *id* field.

| Event Type | Description |
|---|---|
| WINDOW_DESTROY | This a command for the Window object to destroy itself. To destroy a Window, use Window.dispose(). |
| WINDOW_ICONIFY | The Window has been iconified. |
| WINDOW_DEICONIFY | The iconified Window has been restored. |
| WINDOW_MOVED | The Window has been moved on the desktop. |
| KEY_PRESS | The user has pressed a key. Examine *key* and *modifiers* members to see which key. |
| KEY_RELEASE | A pressed key has been released. Examine *key* and *modifiers* members to see which key. |
| MOUSE_DOWN | The mouse button has been clicked. Examine *x* and *y* members to see where the mouse click occurred. |
| MOUSE_UP | The mouse button has been released. Examine *x* and *y* members to see where the mouse click occurred. |
| MOUSE_ENTER | The mouse has entered the rectangle dedicated to this Component. Examine *x* and *y* members to see where the mouse click occurred. |
| MOUSE_EXIT | The mouse has left the rectangle dedicated to this Component. Examine *x* and *y* members to see where the mouse click occurred. |
| MOUSE_DRAG | Same as a MOUSE_MOVE, but with the mouse button held down. |
| MOUSE_MOVE | The mouse has been moved. Examine x and y members to see where the mouse cursor was moved to. |
| SCROLL_LINE_UP | The *target* member holds a reference to the Scrollbar which the user clicked. The *arg* member holds the value of the Scrollbar. |
| SCROLL_LINE_DOWN | The *target* member holds a reference to the Scrollbar which the user clicked. The *arg* member holds the value of the Scrollbar. |
| SCOLL_PAGE_UP | The *target* member holds a reference to the Scrollbar which the user clicked. The *arg* member holds the value of the Scrollbar. |
| SCROLL_PAGE_DOWN | The *target* member holds a reference to the Scrollbar which the user clicked. The *arg* member holds the value of the Scrollbar. |
| SCROLL_ABSOLUTE | The *target* member holds a reference to the Scrollbar which |

| | |
|---|---|
| | the user clicked. The *arg* member holds the value of the Scrollbar. This message is sent when the user holds down and drags the thumb of a Scrollbar. |
| LIST_SELECT | The *target* member holds a reference to the List which the user clicked. The *arg* member holds the value of the selected list item. |
| LIST_DESELECTED | The *target* member holds a reference to the List which the user clicked. The *arg* member holds the value of the selected list item. |
| ACTION_EVENT | When a Button is clicked, *target* is the Button, *arg* is the Button's text. When a menu item was selected, *target* is the selected menu item, *arg* is the Button's text. |
| GET_FOCUS | The Component in *target* has just received focus. |
| LOST_FOCUS | The Component in *target* has just lost focus. |

The *modifiers* member is a bitmap representing the state of special function keys during keyboard and mouse Events. The following table lists the recognized values which may be present in the *modifiers* member. These values are ORed together bitwise to form the values of the *modifiers* member.

| **Mask** | **Description** |
|---|---|
| SHIFT_MASK | Set if the SHIFT key was held down. |
| CTRL_MASK | Set if the CTRL key was held down. |
| META_MASK | Set if the META key was held down. |
| ALT_MASK | Set if the ALT key was held down. |

## translate

**ClassName**

Event

**Purpose**

Changes the *x* and *y* member variables of the Event by some value.

**Syntax**

public void translate(int dX, int dY);

**Parameters**

***int dX, dY***

The Event's *x* member variable is modified by adding *dX* to it, and the Event's *y* member variable is modified by adding *dY* to it.

**Description**

Similar to translate, this method modifies the Event's *x* and *y* member variables by adding a *dX* and a *dY* value to them. Internally, this method is called by Component.postEvent when the Event is passed on to the Component's Container in order to reflect the point in terms of the Container's origin.

**Imports**

None.

**Returns**

None.

**Example**

This Component's overridden mouseDown and mouseUp Event handlers change the location of mouse clicks and mouse releases by ten points in the X and Y directions before allowing the Events to be passed on to the Component's parent Container.

```
public class MyComponent extends Canvas {
    …

    public boolean mouseDown(Event evt, int x, int y) {
        evt.translate(10, 10);
        return false;
    }

    public boolean mouseUp(Event evt, int x, int y) {
        evt.translate(10, 10);
        return false;
    }

    …
}
```

## shiftDown

**ClassName**

Event

**Purpose**

Tells whether or not the SHIFT key was held down when the Event was created.

**Syntax**

public boolean shiftDown();

**Parameters**

None.

**Description**

Tells whether or not the SHIFT_MASK flag is set in the *modifiers* member variable. Using this member method is a little easier than testing the *modifiers* variable directly. The SHIFT, CTRL, and META masks are only valid for keyboard and mouse Events.

**Returns**

True is returned if the SHIFT_MASK flag is set in the *modifiers* member variable. Otherwise, false.

**See Also**

The controlDown and metaDown methods of the Event class

**Example**

This example mouseDown Event handler sends mouse Events to subhandlers according to the states of the SHIFT, CTRL, and META flags.

```
public class MyComponent extends Canvas {
    …

    public boolean mouseDown(Event evt, int x, int y) {
        if(evt.shiftDown())
            return mouseDownShift(evt, x, y);
        if(evt.controlDown())
            return mouseDownCtrl(evt, x, y);
        if(evt.metaDown())
            return mouseDownMeta(evt, x, y);
        return false;
    }

    public mouseDownShift(Event evt, int x, int y) {
        // Do something with mouse clicks while SHIFT is
        // down.
        return true;
    }

    public mouseDownCtrl(Event evt, int x, int y) {
        // Do something with mouse clicks while CTRL is
        // down.
        return true;
    }

    public mouseDownMeta(Event evt, int x, int y) {
        // Do something with mouse clicks while META is
        // down.
        return true;
    }

    …
}
```

## controlDown

**ClassName**

Event

**Purpose**

Tells whether or not the CTRL key was held down when the Event was created.

**Syntax**

public boolean controlDown();

**Parameters**

None.

**Description**

Tells whether or not the CTRL_MASK flag is set in the *modifiers* member variable. Using this member method is a little easier than testing the *modifiers*

variable directly. The SHIFT, CTRL, and META masks are only valid for keyboard and mouse Events.

**Returns**

True is returned if the CTRL_MASK flag is set in the *modifiers* member variable; otherwise, false.

**See Also**

The shiftDown and metaDown methods of the Event class

**Example**

See the example under the shiftDown method of the Event class.

## metaDown

**ClassName**

Event

**Purpose**

Tells whether or not the META key was held down when the Event was created.

**Syntax**

public boolean metaDown();

**Parameters**

None.

**Description**

Tells whether or not the META_MASK flag is set in the *modifiers* member variable. Using this member method is a little easier than testing the *modifiers* variable directly. The SHIFT, CTRL, and META masks are only valid for keyboard and mouse Events.

**Returns**

True is returned if the META_MASK flag is set in the *modifiers* member variable; otherwise, false.

**See Also**

The shiftDown and controlDown methods of the Event class

**Example**

See the example under the shiftDown method of the Event class.

## Window

**Purpose**

The Window class is a top-level Container class.

**Syntax**

public class Window extends Container

**Description**

The Window class is a top-level Container class. Window class objects do not have parent Containers. Instead, Window objects can be thought of as children of the desktop. The Frame and Dialog classes are special types of Window classes. Figure 3-8 shows the class hierarchy of the Window class. Do not create a

Window object directly, but instead use either the Frame class or the Dialog class to create top-level windows. The Window class implements the methods that are shared between the two specific classes.

**PackageName**

*java.awt*

**Imports**

*java.awt.peer.WindowPeer*

**Constructors**

public Window();
public Window(Frame parent);
The first constructor creates a top-level Frame window. The second constructor creates a top-level window which is a child of the passed Frame, such as a modeless Dialog.

**Parameters**

None.

**Example**

See the examples for the Frame and Dialog classes.

**Figure 3-8** The class hierarchy of the Window class

## toBack

**ClassName**

Window

**Purpose**

Sends the Window to the back of the desktop z-order..

**Syntax**

public void toBack();

**Parameters**

None.

**Description**

Sends the Window to the back of the desktop z-order. If the Window is not showing, this call is ignored. The Window automatically loses keyboard focus after this call is made if the window, or any of its children, have the keyboard focus when the call is made.

**Imports**

None.

**Returns**

None.

**See Also**

The toFront method of the Window class

**Example**

In this example, a custom Event is delivered to the MyWindow class to cause it to be sent to the back or the front of the z-order of the top-level windows.

```
public class MyWindow extends Window {
    public static final CUSTOM_TO_BACK = -1;
    public static final CUSTOM_TO_FRONT = -2;

    public MyWindow() {}

    public boolean handleEvent(Event evt) {
        if(CUSTOM_TO_BACK == evt.id)
            toBack();
        else if(CUSTOM_TO_FRONT == evt.id) {
            toFront();
        else
            return super.handleEvent(evt);
        return true;
    }
}
```

## toFront

**ClassName**

Window

**Purpose**

Brings the Window to the front of the desktop z-order..

**Syntax**

public void toFront();

**Parameters**

None.

**Description**

Brings the Window to the front of the desktop z-order. If the Window is not showing, this call is ignored. The Window automatically gains keyboard focus after this call is made.

**Imports**

None.

**Returns**

None.

**See Also**

The toBack method of the Window class

**Example**

See the example for the toBack method of the Window class.

## dispose

**ClassName**

Window

**Purpose**

Destroys the Window object's native windowing system peer.

**Syntax**

public void dispose();

**Parameters**

None.

**Description**

Destroys the Window's native windowing system peer object. Top-level windows must explicitly destroy (dispose) of their peers. Most commonly used when an application's main window receives a WINDOW_DESTROY Event.

**Imports**

None.

**Returns**

None.

**Example**

This example Event handler, for an application's main Frame window, calls dispose when it receives a WINDOW_DESTROY Event.

```
public class MyAppMainFrame extends Frame {
    …

    public boolean handleEvent(Event evt) {
        if(Event.WINDOW_DESTROY == evt.id) {
            dispose();
            return true;
        }

        return super.handleEvent(evt);
    }
    …

}
```

## getWarningString

**ClassName**

Window

**Purpose**

Gets the Applet warning string to display in Frame windows created by Applets.

**Syntax**

public final String getWarningString();

**Parameters**

None.

**Description**

The warning string is a string that displays in a Frame window created by an Applet object. For example, the Netscape Navigator v2.0 displays a string "Untrusted Applet Window" on every Frame window created by Applets. Note that this method is final, so your Applet cannot override this implementation. The warning string is actually a System property called "awt.appletWarning".

**Imports**

None.

**Returns**

A String object containing the warning string to display.

**Example**

This custom Frame class uses the warning string as the Frame's caption.

```
public class MyFrame extends Frame {
    public MyFrame() {
        super(getWarningString());
    }
}
```

## Toolkit

**Purpose**

The Toolkit class represents the native windowing system in Java.

**Syntax**

public abstract class Toolkit

**Description**

The Toolkit class represents the native windowing system in Java. The four functionalities accessible through the Toolkit class are: Component peer creation, Font enumeration and metrics, Screen sizing and resolution, and Image loading and preparation. Figure 3-9 shows the class hierarchy of the Toolkit class. The Toolkit class is an abstract class, so you cannot create an instance of this class. Instead, you use the Toolkit class' getDefaultToolkit method to obtain a reference to the Toolkit implementation in use on the system currently, as demonstrated in the example for the getDefaultToolkit method listed below.

**PackageName**

*java.awt*

**Imports**

*java.awt.peer.*, java.awt.image.ImageObserver, java.awt.image.ImageProducer, java.awt.image.ColorModel, java.net.URL*

**Constructors**

None.

**Parameters**

None.

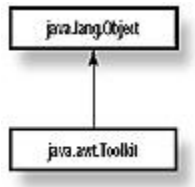


**Figure 3-9** The class hierarchy of the Toolkit class

## getScreenSize

**ClassName**

Toolkit

**Purpose**

Gets the dimension of the desktop in pixels.

**Syntax**

public Dimension getScreenSize();

**Parameters**

None.

**Description**

Gets the dimension of the desktop in pixels. This is very useful for applications which would like to layout their Components based on available on-screen real estate.

**Imports**

*java.awt.Dimension*

**Returns**

The return Dimension object's *width* and *height* members reflect the width and height of the desktop.

**See Also**

The getScreenResolution method of the Toolkit class

**Example**

This example method centers a Frame on the desktop.

```
public void centerFrameOnDesktop(Frame f) {
    Dimension dim = f.size();
    Dimension dimDesktop =
            Toolkit.getDefaultToolkit().getScreenSize();
    f.move( dimDesktop.x / 2 + dim.x / 2,
            dimDesktop.y / 2 + dim.y / 2);
}
```

## getScreenResolution

**ClassName**

Toolkit

**Purpose**

Gets the resolution of the desktop.

**Syntax**

public int getScreenResolution();

**Parameters**

None.

**Description**

Gets the resolution of the desktop in pixels per inch. The number returned is valid in both the X and Y directions. This is useful for applications which need to know physical, not logical, distances on the desktop. For example, an application which is supposed to display a 12-inch ruler would need to know how many pixels from the upper-left corner of the screen is exactly one inch.

**Imports**

None.

**Returns**

The screen resolution in pixels-per-inch. The returned value is valid in both the X and Y directions.

**See Also**

The getScreenSize method of the Toolkit class

**Example**

This example method creates a Frame window which is exactly five inches wide by five inches tall.

```
public Frame make5by5Frame() {
    Frame f = new Frame();
    int pixelsPerInch =
        Toolkit.getDefaultToolkit().getScreenResolution();
    f.resize(5 * pixelsPerInch, 5 * pixelsPerInch );
    f.show();
    return f;
}
```

## getColorModel

**ClassName**

Toolkit

**Purpose**

Gets the ColorModel describing the color capabilities of the desktop.

**Syntax**

public ColorModel getColorModel();

**Parameters**

None.

**Description**

Returns the ColorModel object for the desktop. The ColorModel describes the color palette or color capabilities of the desktop.

**Imports**

*java.awt.image.ColorModel*

**Returns**

A ColorModel object describing the color capabilities of the desktop.

**See Also**

The ColorModel class

**Example**

This example method profiles the number of colors the desktop is capable of displaying simultaneously.

```
public void displaySimulColors() {
    ColorModel cmDesktop =
            Toolkit.getDefaultToolkit().getColorModel();
    int nColorBits = cmDesktop.getPixelSize();
    System.out.println("Max desktop colors: " +
            (1 << nColorBits));
}
```

## getFontList

**ClassName**

Toolkit

**Purpose**

Lists all the font face names available for rendering text.

**Syntax**

public abstract String[] getFontList();

**Parameters**

None.

**Description**

Use this method to get a list of all available fonts on the system. An array is returned, each element of which is a String containing a valid font face name. Use Font.getFont with the font face name to create a Font object for a particular face name.

**Imports**

None.

**Returns**

None.

**Example**

See the example Project for this chapter, FontLab, which uses getFontList to enumerate all font face names available on the local system.

## getFontMetrics

**ClassName**

Toolkit

**Purpose**

Gets the FontMetrics for a particular Font as rendered on the desktop.

**Syntax**

public abstract FontMetrics getFontMetrics(Font font);

**Parameters**

***Font font***

The Font you want to gather metrics for.

**Description**

This method returns a FontMetrics object describing the metrics of a particular Font when rendered on the desktop. The Component.getFontMetrics method is actually a shallow wrapper around this method.

**Imports**

*java.awt.FontMetrics, java.awt.Font*

**Returns**

A FontMetrics object describing the metrics of Font *font* when rendered on the desktop.

**Example**

This example method returns the length, in pixels, of a String when displayed on the desktop using a particular Font. The Font is described only by a font face name, such as one of the elements returned by Toolkit.getFontList.

```
public int getStringWidthInFont(String str, String strFaceName) {
    Font font = Font.getFont(strFaceName);
    FontMetrics fm =
```

```
        Toolkit.getDefaultToolkit().getFontMetrics(font);
    return fm.stringWidth(str);
}
```

## getDefaultToolkit

**ClassName**

Toolkit

**Purpose**

Gets the Toolkit object used by the AWT.

**Syntax**

public static synchronized Toolkit getDefaultToolkit();

**Parameters**

None.

**Description**

Gets the Toolkit object used by the *java.awt.** packages. There is nothing stopping you from implementing another Toolkit in addition to the default Toolkit. For example, if you wanted to take advantage of some native windowing system capabilities, which are not available through Java's Toolkit object, you could implement your own and use it instead of Java's Toolkit object.

**Imports**

None.

**Returns**

The Toolkit object used within the Java API classes is returned.

**Example**

See the examples for the methods getFontMetrics and getScreenResolution in the Toolkit class.

## getImage

**ClassName**

Toolkit

**Purpose**

To load an image from a URL and prepare it for rendering on the desktop.

**Syntax**

public abstract Image getImage( String filename );public abstract Image getImage( URL url );

**Parameters**

***String filename***

The full path name for a graphical format file on the local file system.

***URL url***

Points to an image file to be loaded by the Toolkit.

**Description**

This method allows any code in Java to initiate loading of an Image from a graphical image file. This graphical format file may be a location in a file on the local file system, or is indicated by a URL (available or the Internet). The first

overloaded version of this method loads images from files on the local file system, and the second loads images from files available over the Internet.

**Imports**

*java.awt.Image*

**Returns**

An Image object will be returned by this object. The reaction of this method when the URL refers to an unsupported protocol or when the image file format is unrecognized or is unspecified. Generally, it can be assumed that null will be returned if this capability is not provided by the Toolkit.

**See Also**

The Image class

**Example**

The following sample Component loads and displays an image. A relative URL to the image to be loaded is passed to the Component's constructor. The Component acts as the ImageObserver for the Image construction process.

```
public class ImageComp extends Component {
  Image image;

  public ImageComp(URL urlImage) {
    image = Toolkit.getDefaultToolkit().getImage(urlImage);

  public void paint( Graphics g ) {
    // Paint image on display surface, if image exists
    if( null != image )
        g.drawImage( image, 0, 0, this );
  }
}
```

## prepareImage

**ClassName**

Toolkit

**Purpose**

Kick-starts the Image construction process for an Image to be displayed with a specified width and height.

**Syntax**

public boolean prepareImage(Image img, int width, int height, ImageObserver observer);

**Parameters**

***Image img***

The Image object to create a screen representation of.

***int width***

***int height***

The scaled size of the Image's representation.

***ImageObserver observer***

The ImageObserver object to receive notification of the asynchronous progress of the construction of the Image's representation.

**Description**

Starts construction of a screen representation of an Image object. An Image must be constructed before it can be displayed on a Component's surface. Note that when you use Graphics.drawImage with a reference to an unconstructed Image object, the Image's construction process is automatically started for you. The prepareImage method allows you to start this process before the Image is displayed on any surface.

**Imports**

*java.awt.Image, java.awt.image.ImageObserver*

**Returns**

True is returned if the representation of the Image object is complete. Otherwise, false is returned and the Image construction process is started.

**See Also**

The Image class; the ImageObserver interface; the checkImage method of the Toolkit class

**Example**

See the example for the checkImage method of the Toolkit class.

## checkImage

**ClassName**

Toolkit

**Purpose**

To check the status of construction of an Image.

**Syntax**

public int checkImage(Image img, int width, int height, ImageObserver observer);

**Parameters**

***Image img***

The Image object whose status is to be checked.

***int width***

***int height***

The scaled size of the image representation being checked.

***ImageObserver observer***

An ImageObserver object currently being notified of the progress of construction of the Image object.

**Description**

Checks the status of the construction of an Image object. The ImageObserver is continuously notified about the progress of the image construction process through its updateImage method. checkImage allows non-ImageObserver objects to poll for the progress of this process.

**Imports**

None.

**Returns**

A logical ORing of the ImageObserver flags indicating what information about the Image is available. This can include one or more of the following ImageObserver values: WIDTH, HEIGHT, PROPERTIES, SOMEBITS, FRAMEBITS, ALLBITS, ERROR.

**See Also**

The ImageObserver interface

**Example**

This example prevents the Component from painting its surface until the Image construction flag ALLBITS has been passed to the ImageObserver watching the image construction process.

```
public MyComponent extends Canvas {
    Image _img;

    // Constructor takes an Image parameter and begins
    // construction of it.
    public MyComponent(Image img) {
        _img = img;
        Toolkit.getDefaultToolkit().
            prepareImage(_img, 100, 100, this);
                    // uses the comp as the ImageObserver.
    }

    // paint does nothing until image has been
    // fully constructed.
    public void paint(Graphics g) {
        if(0 == (ImageObserver.ALLBITS &
                 checkImage(_img, 100, 100, this)))
            return;

        // Do something with the image
        …
    }
}
```

## createImage

**ClassName**

Toolkit

**Purpose**

Creates an in-memory Image from pixel data provided by an ImageProducer.

**Syntax**

public Image createImage(ImageProducer producer);

**Parameters**

***ImageProducer producer***

The ImageProducer object which will provide the data defining the resultant Image.

**Description**

The resultant Image will have a compatible ColorModel to the display device associated with this Component object. This method creates the Image using pixel data provided by the ImageProducer. (See Chapter 8, which describes image processing methods and techniques in Java.)

**Imports**

*java.awt.image.ImageProducer*

**Returns**

An Image object.

**See Also**

The ImageProducer class

**Example**

This example uses createImage along with a fictitious FakeFilter, which is supposed to be any type of ImageFilter (See Chapter 8 for a discussion of ImageProducers, ImageConsumers, and ImageFilters).

```
// Assume a URL has been provided for the
// source Image…

Image imageSource =
    Toolkit.getDefaultToolkit().getImage(urlSource);
Image imgFiltered =
    Toolkit.getDefaultToolkit().createImage(
        new FilteredImageSource(imgSource.getSource(),
                                new FakeFilter())
    );
```

## The Project: FontLab

The SuperBible Project for this chapter is called FontLab. FontLab is a relatively simple Java application that illustrates the use of z-ordering to arrange Components within a Container. All of the FontLab classes are defined within the same .JAVA file, FontLab.java. The Project can be found on the CD that accompanies this book in the directory \WHERE\THE\PROJECT\IS.

Figure 3-10 shows the FontLab application running. One thing you may notice right away about the FontLab interface is that there are several overlapping panels in the main window. That's one of the lessons of FontLab: how to make pseudo-MDI (multi-document interface) applications. Another lesson is z-ordering of child Components.


**Figure 3-10** Screenshot of the FontLab project

### Assembling the Project

**1.** Create a file named FontLab.java. This file will hold all the code for this project.

**2.** The first class to create is the application class, which will implement our static main() method. Also, ensure the proper packages are imported. The code for this step is

```
import java.awt.*;

public class FontLab {
```

```
    public static void main(String[] astrArgs) {
        FontLabFrame f = new FontLabFrame("Font Lab");
        f.resize(700, 500);
        f.show();
    }
}
```

**3.** Create our application's main Frame class. This Frame contains the various panels to display each Font. The constructor creates each of the display panels and adds them. The code for this step is

```
class FontLabFrame extends Frame {
    FontDisplay[] aDisplays;

    public FontLabFrame(String strTitle) {
        super(strTitle);

        // Set background color and NullLayoutManager.
        setBackground( Color.white );
        setLayout(new NullLayout());

        // Get the list of available fonts.
        String[]astrFonts =
Toolkit.getDefaultToolkit().getFontList();

        // Create a display for each typeface name.
        aDisplays = new FontDisplay[astrFonts.length];
        int cxInc = 200/astrFonts.length;
        int cyInc = 200/astrFonts.length;

        for( int ii=astrFonts.length-1 ; ii>0 ; ii-- ) {
            aDisplays[ii] = new FontDisplay(astrFonts[ii]);
            aDisplays[ii].reshape(ii*cxInc, ii*cyInc, 500, 300);
            add(aDisplays[ii]);
        }
    }
}
```

**4.** As with all Frame windows, the FontLabFrame class must handle all WINDOW_DESTROY Events to dispose of the window. The code for the handleEvent method of the FontLabFrame class is

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

**5.** When a mouse click occurs anywhere in the application, FontLabFrame should handle it. When handling a mouse click, locate the child Component on which the click occurred, then remove and re-add the associated FontDisplay panel so it is brought to the proper z-order.

```
public boolean mouseDown(Event evt, int x, int y) {
    Component c = (Component)evt.target;
    while(!((null==c) || (c instanceof FontDisplay)))
        c = c.getParent();
```

```
    if(null != c) {
        remove(c);
        add(c );
        validate();
        return true;
    }
    return false;
}
```

**6.** A FontDisplay panel displays sample text using a particular Font. The Font is given to the FontDisplay object's constructor when it is created. A toolbar is also provided so the user may change the sample text and the size it is displayed at. Creation of the toolbar and initialization of the FontDisplay's member occurs in the constructor. Here is the code:

```
class FontDisplay extends Panel {
    Insets _insets = new Insets(5, 5, 5, 5);
    String _strFontName;
    TextField _textString = new TextField("Sample", 15);
    TextField _textSize = new TextField("10", 2);

    public FontDisplay(String strFontName) {
        _strFontName = strFontName;

        // Set background color and BorderLayout.
        setBackground( Color.gray );
        setLayout(new BorderLayout());

        // Create font.
        setFont(new Font(_strFontName, Font.PLAIN, 10));

        // Create title bar.
        Panel panelTitle = new Panel();
        panelTitle.setLayout(new FlowLayout());
        Label labelTitle = new Label(_strFontName);
        panelTitle.add(labelTitle);
        panelTitle.setFont(getFont());
        add("North", panelTitle);

        // Create font selection bar at the bottom
        Panel panelChoose = new Panel();
        panelChoose.setLayout(new FlowLayout(FlowLayout.LEFT, 5,
5));
        Label labelString = new Label("String:");
        Label labelSize = new Label("Height:");
        Button buttonUpdate = new Button("Update");
        panelChoose.add(labelString);
        panelChoose.add(_textString);
        panelChoose.add(labelSize);
        panelChoose.add(_textSize);
        panelChoose.add(buttonUpdate);
        panelChoose.setFont(getFont());
        add("South", panelChoose);
    }
}
```

**7.** Add an inset around the FontDisplay so child Components aren't butted up against the FontDisplay's borders. Here is the code:

```
// Return insets of 5 in all directions
public Insets insets() {
    return _insets;
}
```

**8.** The paint method of the FontDisplay class simply draws the sample text using the indicated Font. The sample text is centered within the FontDisplay object. Here is the code:

```
public void paint(Graphics g) {
        Rectangle r = bounds();
        r.move(0, 0);
        g.drawRect(r.x, r.y, r.width-1, r.height-1);

        // Draw the string centered.
        String strText = _textString.getText();
        FontMetrics fm = g.getFontMetrics();
        g.drawString(strText, r.width/2 -
fm.stringWidth(strText)/2,
                r.height/2);
    }
```

**9.** When the user hits the Update button on the FontDisplay, the FontDisplay receives an ACTION_EVENT, handled by the action method. Our implementation reads in the new sample text and Font size, and updates the FontDisplay's member variables accordingly. Here is the code:

```
    // When the Update button is hit, change to new font
    // size and repaint.
    public boolean action(Event evt, Object what) {
        // Make sure 'what' is a string.
        if(!(what instanceof String))
            return false;

        // If it isn't the Update button, ignore.
        if(!"Update".equals((String)what))
            return false;

        // Get the new font size and create a new font.
        String strSize = _textSize.getText();
        int nSize;
        try {
            nSize = Integer.parseInt(strSize);
        } catch (Exception e) {
            return false;
        }

        setFont(new Font(_strFontName, Font.PLAIN, nSize));
        return true;
    }
```

**10.** Finally, implement the NullLayout class, which is a LayoutManager that essentially does nothing. This allows us to place the FontLabFrame's child Components (FontDisplay objects) in overlapping positions. Chapter 6 discusses LayoutManagers. Here is the code:

```
// The NullLayout is a no-op layout manager. It just leaves
// all Components in the target alone wherever they have
// been placed through Component.reshape(), move() or
// resize() calls.
```

```
class NullLayout implements LayoutManager {
    public void addLayoutComponent(String name,
            Component c) {}
    public void removeLayoutComponent(
            Component c) {}
    public Dimension preferredLayoutSize(Container target) {
        return new Dimension(0, 0);
    }
    public Dimension minimumLayoutSize(Container target) {
        return new Dimension(0, 0);
    }
    public void layoutContainer(Container target) {}
}
```

## How It Works

### *The FontLab Classes*

Within FontLab, there are four predefined classes. Tabele 3-3 lists the four classes and a description of each.

**Table 3-3** Listing the classes of the FontLab project application

| Class | Description |
|---|---|
| FontLab | The application class, which defines the static main() method. The only task of the main() method is to create and resize the FontLab main window. The FontLab main window is of the FontLabFrame class. The FontLab class is derived from nothing. |
| FontLabFrame | There is only a single instance of the FontLabFrame class in each instance of the FontLab application. This instance is the main frame window of the Application. The frame window is responsible for creating and placing the FontDisplay panels, one for each of the typefaces available on the system. The FontLabFrame must also end the application by disposing of itself when it receives a WINDOW_DESTROY Event. FontLabFrame is derived from the Frame class. |
| FontDisplay | There is a single instance of FontDisplay for each of the typefaces available on the system. FontDisplay is derived from Panel. Looking at Figure 3-10, the FontDisplay objects are the four overlapping cards in the middle of the main window. The task of a single FontDisplay object is to display a line of text, specified by the user, in a particular font with a particular size; the size is also specified by the user. |

| | |
|---|---|
| NullLayout | The NullLayout is a LayoutManager which basically doesn't do anything. It allows the child Components of the Container to just sit wherever their own move() and reshape() methods have placed them. In FontLab, which has several overlapping FontDisplay panels within the main window, the main window uses a NullLayout instance to (not) manage its child Components. |

### *During Program Initialization*

When FontLab starts up, the FontLab.main() method is run. The main() method only has three lines of code:

```
FontLabFrame f = new FontLabFrame("Font Lab");
f.resize(700, 500);
f.show();
```

That is, it creates the main window, resizes it to a predefined size, shows the main frame, and quits.

Most of the initialization work is done within the FontLabFrame's constructor. That constructor has two tasks: First, it gets the list of available fonts from the Toolkit, using Toolkit.getFontList. Second, it creates a FontDisplay object for each of the available fonts, sizes it, places it, and adds it as a Component of the main frame window. Here's the code from the FontLabFrame constructor that performs those steps:

```
// Get the list of available fonts.
        String[] astrFonts = Toolkit.getDefaultToolkit().getFontList();

        // Create a display for each typeface name.
        aDisplays = new FontDisplay[astrFonts.length];
        int cxInc = 200/astrFonts.length;
        int cyInc = 200/astrFonts.length;

        for( int ii=astrFonts.length-1 ; ii>0 ; ii-- ) {
            aDisplays[ii] = new FontDisplay(astrFonts[ii]);
            aDisplays[ii].reshape(ii*cxInc, ii*cyInc, 500, 300);
            add(aDisplays[ii]);
        }
```

The sizing of the FontLabFrame and the various FontDisplay panels is hard-coded to keep the code complexity to a minimum. What's not shown above is that the FontLabFrame sets its LayoutManager to a NullLayout object, but that also occurs within the FontLabFrame constructor.

The last step in the initialization process is the FontDisplay constructor, which is used to create each of the FontDisplay panels. As you can see in Figure 3-10, the FontDisplay panel is made up of three parts: The title at the top of the panel, the sample text in the

center of the panel, and a toolbar allowing the user to write in sample text and a text size in the lower panel. The FontDisplay constructor creates these three elements before quiting.

Each FontDisplay panel is supposed to represent one of the available fonts for the system. The FontDisplay constructor takes the typeface name it is supposed to represent as the only argument. The FontDisplay stores this typeface name in a member variable, and sets its font to a 10-point Font based on this typeface name:

```
public FontDisplay(String strFontName) {
        _strFontName = strFontName;

[…]

        // Create font.
        setFont(new Font(_strFontName, Font.PLAIN, 10));
```

Next, the FontDisplay constructor creates its titlebar and toolbar. The focus of this chapter is not on user-interface creation, so we will resist going into detail about that here.

## *Changing Z-Order*

FontLab was designed with overlapping windows to demonstrate z-ordering. When FontLab starts, all of the FontDisplay panels are added to the main frame window in quick succession. If you noticed in the code snippet taken from the FontLabFrame constructor, the constructor actually adds the panels to the frame in the order they are to appear initially, top to bottom. That is, the panel which should end up on the top of the z-order is added first, then the second, and so on. Remember that the component added last is at the bottom of the z-ordering, so when adding several Components at once to a Container, you'll want to add the items in a top-to-bottom order.

FontLab was written so that a user-click on a panel sends the panel to the bottom of the z-order. The code which changes the z-ordering is in FontLabFrame.mouseDown. In mouseDown, the frame determines which panel the mouse was clicked in, if any. That panel is removed from the frame and re-added. This sends the panel to the bottom of the z-order.

Note that while you might at other times use Container.locate to determine which child Component a particular point was in, that would not work for FontLab. Remember that Container.locate does not work when you have overlapping child Components. FontLabFrame.mouseDown() determines which panel the mouse click occurred in by examining the *target* member of the MOUSE_DOWN Event. The target Component will be one of three types: FontDisplay panel, child Component of a FontDisplay panel, or the FontLabFrame itself.

FontLabFrame determines which panel to change the z-order for by examining the ancestors of the *Event.target* member. The first one that is a FontDisplay is the panel the mouse click occurred in. If the code gets to the top of the window hierarchy, then the mouse click was not in a FontDisplay panel, and the mouse click can be ignored. Here's the code:

```
Component c = (Component)evt.target;
while(!((null==c) || (c instanceof FontDisplay)))
    c = c.getParent();

if(null != c) {
    remove(c);
    add(c);
    validate();
    return true;
}
```

# Chapter 4
# Windowing Components And Layout Classes

Building a graphical user interface for an application is an interesting and challenging task. Every graphical user interface requires components that perform specialized tasks, such as buttons that invoke actions and data entry areas for users to type in information. Components also need to be logically grouped. Java's Abstract Windowing Toolkit (AWT) provides a rich set of ready-to-use user interface components that application developers can incorporate into their programs.

This chapter describes the application programming interface of some of the ready-made GUI components that the Java toolkit provides. It also introduces the concepts involved in laying out components on the screen using Java's layout managers. It explains how to use the Java layout managers and how to implement custom layout managers. Throughout, detailed explanations of the methods are supplemented with examples that will assist you in building an attractive GUI. The project, at the end of this chapter, lays out some of the components described in this chapter using layout managers and demonstrates the capabilities of each of Java's layout managers.

## Windowing Components

As a developer of GUIs, you will use Java's GUI components a lot. Some of them, such as Button, Canvas, Frame, Panel, Label, and Scrollbar are common user-interface components that nearly all GUI toolkits provide. Some classes, such as Dimension and Insets, are helper classes that embody abstractions that help make programming in a graphical environment easy. Figure 4-1 shows several components of a GUI.

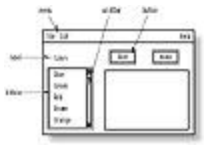

**Figure 4-1** Components of a graphical user interface

## Layouts

All the GUI components in the Java AWT are implemented with subclasses of the Component class. As you may recall from Chapter 3, the Container is a special type of component that can contain other components. How does one arrange the Components within a container? This is where the layout manager plays a role. A layout manager is a Java object that knows how to position and size or resize components within a container that appears on the screen. Java provides a number of layout managers, each of which lays out components differently. Every container has a default layout manager that you can easily replace with another one. You can also specify absolute positions (for components) instead of using a layout manager.

Layout managers must implement the methods of the LayoutManager interface. Since all layout managers implement the same interface, if you know how to use one layout manager, then switching to a different one is easy. Applications do not invoke the methods of the LayoutManager interface directly. The Container methods add, remove, removeAll, layout, preferredSize, and minimumSize result in calls to the corresponding methods of the layout manager associated with that Container object. The layout managers that Java's AWT provides are FlowLayout, GridLayout, BorderLayout, CardLayout, and GridBagLayout. Figure 4-2 shows two examples of what you can do with these layout managers.

**Figure 4-2** Examples of BorderLayout and GridBagLayout

## Summary of Windowing Component and Layout Classes

Table 4-1 summarizes Java's Windowing Component classes. Java's layout interfaces and classes are summarized in Table 4-2.

**Table 4-1** Windowing class descriptions

| Class | Description |
|---|---|
| Button | A class that implements a button, that can be pressed to invoke a user-defined action/sequence of actions. |

| | |
|---|---|
| Canvas | This class implements a generic canvas that drawing operations can be performed on, using the specified graphics device for the target device. |
| Dimension | The Dimension class encapsulates the height and width measurement of a component. |
| Frame | A class that represents a top-level window, which can function as a container for other components. |
| Insets | A class that specifies an inset from within a rectangular area. |
| Label | This class implements a graphical component, that displays a single line of noneditable text. |
| Panel | A class that implements a generic container in which other components can be laid out. |
| Scrollbar | A class that represents a scrollbar. |

**Table 4-2** Layout class and interface descriptions

| Class/Interface | Description |
|---|---|
| LayoutManager | This defines an interface that all layout managers must implement in order to be used for laying out Components. |
| FlowLayout | This is a very simple layout manager that lays out components in rows, and is the default layout manager for all Panels. |
| GridLayout | This class implements a layout manager that lays out Components in a grid with a specified number of rows and columns and resizing the Components so that they are all of equal size. |
| BorderLayout | This is the default layout manager for all Windows and configures the layout of the container, using areas named North, South, East, West, and Center. |
| CardLayout | A layout manager for a Container that arranges its Components into cards and stacks the cards, so that only one card is visible at any time. |
| GridBagConstraints | This class specifies the constraints for laying out components using the GridBagLayout manager. |
| GridBagLayout | A very flexible layout manager that lays out components, aligning them vertically and horizontally. A set of constraints specifies how each Component is laid out within its display area. |

## Button

**Purpose**

This class represents an on-screen button.

**Syntax**

public class Button extends Component

**Description**

This class implements a button that can be pressed to invoke a user-defined action or sequence of actions. Methods of the Component class can be used with Button objects. Refer to the Event class, described in Chapter 3, to find out more about handling button related events. Figure 4-3 shows the inheritance hierarchy for the Button class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.Button;*

**Constructors**

public Button()
public Button(String *label*)

**Parameters**

***label***

The text string to display on the button

**Example**

The following sample code shows how to construct Button objects.

```
import java.awt.*;

public class TestButton extends Frame {
    TestButton() {
        super("Testing Button constructors");    // application title
        // Construct buttons
        Button b1 = new Button();
        Button b2 = new Button("Test");
        Button b3 = new Button("Long button label");
        // Add the buttons to the frame window
        add("North", b1);
        add("Center", b2);
        add("South", b3);
        resize(150, 100);
        show();
    }

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

**Figure 4-3** Inheritance hierarchy for the Button class

## addNotify()

**ClassName**
Button
**Purpose**
This method creates a peer object for this Button object.
**Syntax**
public synchronized void addNotify()
**Parameters**
None.
**Description**
This method creates a peer for this Button. The peer allows the programmer to change the look of the button without affecting its functionality. The addNotify method is the earliest stage, in the creation of a component, that platform specific resources (such as color, fonts, and fontmetrics) may be determined. Classes that override this method must first call super.addNotify() before doing any other processing in this method.
**Imports**
*import java.awt.Button;*
**Returns**
None.
**See Also**
The ButtonPeer interface
**Example**
Refer to the example listed in the addNotify method of the Canvas class in this chapter and also to the section on ButtonPeer interfaces in Chapter 9.

## getLabel()

**ClassName**
Button
**Purpose**
To retrieve the text string displayed on the button.
**Syntax**
public String getLabel()
**Parameters**
None.
**Description**

This method retrieves the text label displayed on the button.

**Imports**

*import java.awt.Button;*

**Returns**

The return type of this method is String. This return value contains the text string displayed on the Button object.

**See Also**

The setLabel method

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestButton extends Frame {
     TestButton() {
          super("Testing Button.getLabel()");  // Application title
          Button b = new Button("Howdy!");   // Construct button
          add("Center", b);                    // Add the button to the
                                                    frame window
          // print the label of the button on the standard output
device
                                                          terminal)
          System.out.println("Button label: " + b.getLabel());
          resize(150, 100);
          show();
       }

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

## paramString()

**ClassName**

Button

**Purpose**

To represent the parameters of this Button as a String object.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method represents the various parameters associated with the Button as a String and returns the string. Classes that extend the Button class can override this method to add additional parameter information to the String representation. This protected method cannot be invoked from an application, but is invoked by the toString method of the Component class.

**Imports**

*import java.awt.Button;*

**Returns**

The return type of this method is String. This return value contains the text label of the Button object, in addition to the parameter values of the base class, Component.

**See Also**

The toString and paramString methods of the Component class.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestButton extends Frame {
    TestButton() {
        super("Testing Button.paramString()");     // Application title
        Button b = new Button("Howdy!");           // Construct button
        add("Center", b);       // Add the button to the frame window
        // print the parameters associated with this Button
        System.out.println("Button label: " + b.toString());
        resize(150, 100);
        show();
    }

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

## setLabel(String)

**ClassName**

Button

**Purpose**

To set the label on the button to the specified string.

**Syntax**

public void setLabel(String label)

**Parameters**

***label***

The text string for the Button label.

**Description**

This method changes the text label, that is displayed on button, to the specified string.

**Imports**

*import java.awt.Button;*

**Returns**

None.

**See Also**

The getLabel method

**Example**

The following code illustrates the use of this method in setting the label of a button.

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

```
public class TestButton extends Frame {
  TestButton() {
     super("Testing Button.setLabel()");  // Application title
     Button b = new Button();          // Construct a button with no
label
     b.setLabel("Quit");               // specify the label text
     add("Center", b);                 // Add the button to the frame
          window
     resize(150, 100);
     show();
  }

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

## Canvas

**Purpose**

A generic canvas type on which graphics operations such as drawing can be performed.

**Syntax**

public class Canvas extends Component

**Description**

This class implements a generic canvas type, on which drawing operations can be performed using the specified graphics device. Classes that extend the Canvas class must override the minimumSize method, as the default size of a Canvas is zero. Figure 4-4 shows the inheritance hierarchy for the Canvas class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.Canvas*;

**Constructors**

public Canvas()

**Parameters**

None.

**Example**

The following sample code shows how to construct Canvas objects.

```
import java.awt.*;

public class TestCanvas extends Frame {
    TestCanvas() {
        super("Canvas test");      // Application title
        Canvas c = new Canvas();   // Construct a canvas object
        add("Center", c);          // add the canvas to the Frame
window
        resize(150, 200);
        show();
```

```
    }

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

**Figure 4-4** Inheritance hierarchy for the Canvas class

## addNotify()

**ClassName**

Canvas

**Purpose**

To create a peer object for this Canvas object.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

This method creates a peer for this Canvas object that enables you to change the appearance of the canvas without changing the functionality of the Canvas object. The addNotify method is the earliest stage in the creation of a component where platform specific resources (such as color, fonts, and fontmetrics) may be determined. Classes that override this method *must* first call super.addNotify() before doing any other processing in this method.

**Imports**

*import java.awt.Canvas*;

**Returns**

None.

**See Also**

The interface CanvasPeer

**Example**

The following code illustrates the use of this method. The ColorCanvas class initializes the font type and the text color in the addNotify method and uses these values while painting the Canvas.

```
import java.awt.*;
class ColorCanvas extends Canvas {
    Font canvasFont_;        // The font to use on this canvas
    Color textColor_;        // The color to use for text on this
canvas
    public Dimension minimumSize() {
        return new Dimension(150, 150);
```

```
    }

    public void addNotify() {
        super.addNotify();   // always call super's addNotify first
        textColor_ = new Color(250, 0, 0);    // Red:250 Green:0 Blue:0
        canvasFont_ = new Font("courier", Font.BOLD, 16);
    }

    public void paint(Graphics g) {
        g.setColor(textColor_);         // set the text color
        g.setFont(canvasFont_);         // set the font
        g.drawString("Custom Canvas", 5, 25); // write a string
    }
}

public class TestCanvas extends Frame {
    TestCanvas() {
        super("Another Canvas demo");         // Application title
        ColorCanvas c = new ColorCanvas();    // Construct the color
                                                  canvas object
        add("Center", c);                   // add the canvas to the
Frame
                                                  window
        resize(150, 200);
        show();
    }

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

## paint(Graphics)

**ClassName**

Canvas

**Purpose**

To paint the Canvas object.

**Syntax**

public void paint(Graphics g)

**Parameters**

***g***

The graphics context object.

**Description**

This method is invoked to paint the canvas object. Classes that extend the Canvas class can customize the appearance of the canvas by overriding this method.

**Imports**

*import java.awt.Canvas*;

**Returns**

None.

**See Also**

The Component and Graphics classes

**Example**

This sample source code implements a custom canvas type that has horizontal lines, just like a ruled sheet of paper.

```
import java.awt.*;

public class TestCanvas extends Frame {
    TestCanvas() {
super("Canvas test");                       // Application title
        add("Center", new RuledCanvas());  // create and add the custom
                                             canvas to the frame
        resize(150, 200);
        show();
    }

    public static void main(String args[]) {
        TestCanvas t = new TestCanvas();
    }
}

// This class implements a ruled canvas type
class RuledCanvas extends Canvas {
    public void paint(Graphics g) {
        Rectangle r = bounds();
        g.setColor(Color.black);           // set the line color to
black
        // draw ruled lines across the width of the canvas
        for (int i = 1; i <= r.height/10; i++) {
            g.drawLine(0, i * 10, r.width, i * 10);
        }
    }
}
```

## Dimension

**Purpose**

A class that represents a height and a width measurement.

**Syntax**

public class Dimension extends Object

**Description**

The Dimension class encapsulates the height and width measurement of a component. The width and height variables of a Dimension object are public and, hence, can be accessed directly. Figure 4-5 shows the inheritance hierarchy for the Dimension class.

**PackageName**

*java.awt*

**Imports**

*import import java.awt.Dimension;*

**Constructors**

public Dimension()
public Dimension(Dimension *d*)
public Dimension(int *width*, int *height*)

**Parameters**

***d***

The source Dimension object from which to copy.

***width***

The width measurement of the dimension.

***height***

The height measurement of the dimension.

**Example**

This code illustrates the different ways of constructing Dimension objects.

```
import java.awt.Dimension;

public class DimensionTest {
    public static void main(String args[]) {
      // Dimension constructors
      Dimension d1 = new Dimension();          // void constructor
      Dimension d2 = new Dimension(100, 200); // width and height
                                                    specified
      Dimension d3 = new Dimension(d2);        // construct a copy of d2
      Dimension d4 = new Dimension();
      d4.width = 25;                           // set the width and
height
      d4.height = d3.height + 100;             // individually
    }
}
```


**Figure 4-5** Inheritance hierarchy for the Dimension class

## toString()

**ClassName**

Dimension

**Purpose**

To represent the parameter values of the Dimension object as a String.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method returns, as a string, the values of the width and height measurements of this Dimension object prefixed by its classname (java.awt.Dimension).

**Imports**

*import java.awt.Dimension;*

**Returns**

None.

**See Also**

The toString method of the Object class

**Example**

```
import java.awt.Dimension;

public class DimensionTest {
    public static void main(String args[]) {
        // Dimension constructor specifying width and height
        Dimension d = new Dimension(100, 200);
        System.out.println("d.toString() = " + d.toString());
    }
}
```

When this example is compiled and executed, the following string is printed on the screen.

```
d.toString() = java.awt.Dimension[width=100,height=200]
```

## Frame

**Purpose**

A class that represents a top-level window that can function as a container for other components.

**Syntax**

public class Frame extends Window implements MenuContainer

**Description**

This class implements a window with a title bar and border that can contain a menu bar, as well as other AWT Components. Frames and Panels are commonly used as the top-level window GUIs. Most of the functionality for this class is implemented in the Window, Container, and Component classes. Refer to those classes in Chapter 2 and Chapter 3 to get a complete picture of what you can do with Frame objects. The default layout manager for Frame window objects is BorderLayout. Figure 4-6 shows the inheritance hierarchy for the Frame class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.Frame;*

**Constructors**

public Frame()

public Frame(String *title*)

**Parameters**

***title***

The string to display in the title bar of the Frame window.

**Example**

This example demonstrates Frame construction.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
        // Construct a Frame and put a title on it
        Frame f = new Frame("Frame windows");
        // Component methods such as setBackground() can invoked be on
a
               Frame
        f.setBackground(Color.blue);
```

```
        f.resize(200, 200);
        f.show();
    }
}
```

**Figure 4-6** Inheritance hierarchy for the Frame class

## addNotify()

**ClassName**
Frame
**Purpose**
Creates a peer object for this Frame object.
**Syntax**
public synchronized void addNotify()
**Parameters**
None.
**Description**
This method creates a peer for this Frame object that enables you to change the appearance of the frame window without changing its functionality. The addNotify method is the earliest stage, in the creation of a component, that platform specific resources (such as color, fonts, and fontmetrics) may be determined. Classes that override this method must first call super.addNotify() before doing any other processing in this method.
**Imports**
*import java.awt.Frame;*
**Returns**
None.
**See Also**
The FramePeer interface
**Example**
Refer to the example listed in the addNotify method of the Canvas class in this chapter and also to the section on FramePeer interfaces in Chapter 9.

## dispose()

**ClassName**
Frame
**Purpose**
To release all the resources that are being used by this Frame object.
**Syntax**
public synchronized void dispose()

**Parameters**

None.

**Description**

This method is called to release the resources of a Frame object that is no longer required.

**Imports**

*import java.awt.Frame;*

**Returns**

None.

**See Also**

The Window class

**Example**

Refer to the corresponding method in the Window class description in Chapter 3.

## getCursorType()

**ClassName**

Frame

**Purpose**

To get the integer constant that represents the cursor type associated with this Frame object.

**Syntax**

public int getCursorType()

**Parameters**

None.

**Description**

This method gets the integer type, associated with the cursor for this Frame. A list of cursor types is given in the description of the setCursor method for this class.

**Imports**

*import java.awt.Frame*;

**Returns**

The cursor type, associated with this Frame window, is returned as an integer value and will be one of the values detailed in the list of cursor types.

**Example**

This example prints the integer value of the default cursor image.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
      Frame f = new Frame("Testing Frame.getCursorType");   //
application
                                                              title
      System.out.println("Frame.getCursorType() = " +
f.getCursorType());
      f.resize(200, 200);
      f.show();
```

```
  }
}
```

## getIconImage()

**ClassName**

Frame

**Purpose**

To get the image of the icon of this Frame object.

**Syntax**

public Image getIconImage()

**Parameters**

None.

**Description**

This method gets the image that is displayed when the Frame is iconized.

**Imports**

*import java.awt.Frame;*

**Returns**

This method returns an Image object that represents the icon image used for this Frame object. Refer to the section on Image objects in Chapter 8 for more information.

**See Also**

The Image class

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
      Frame f = new Frame("Testing Frame.getIconImage()");  //
application
          title
      // create an Image object and assign it to the Frame as the
Frame's
      // icon image
      ......

      f.resize(200, 200);
      f.show();
  }
  // this method is invoked to modify the icon being displayed for this
       image
  protected void modifyIcon(Frame f) {
      Image im = f.geIconImage();       //get the image object
      // modify the icon image
      ......
  }
}
```

## getMenuBar()

**ClassName**

Frame

**Purpose**

To get the menu bar object associated with this Frame object.

**Syntax**

public MenuBar getMenuBar()

**Parameters**

None.

**Description**

This method gets the MenuBar object that represents the menu bar associated with this Frame.

**Imports**

*import java.awt.Frame;*

**Returns**

The menu bar information, associated with this Frame window, is returned as a MenuBar object. Refer to the section on MenuBar objects in Chapter 6 for more information.

**See Also**

The MenuBar class

**Example**

The portion of code given here uses this method.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
        Frame f = new Frame("Testing Frame.getMenuBar()");  //
application
             title
        // create a MenuBar object an attach it to the Frame
        ......

        f.resize(200, 200);
        f.show();
    }
    // this method is invoked to modify the menu
    protected void modifyIcon(Frame f) {
        MenuBar mb = f.getMenuBar();      // get the MenuBar object
        // add/delete menu items from the MenuBar
        ......
    }
}
```

## getTitle()

**ClassName**

Frame

**Purpose**

To get the text string on the title bar of the frame.

**Syntax**

public String getTitle()

**Parameters**

None.

**Description**

This method gets the text label displayed in the title bar of the Frame window.

**Imports**

*import java.awt.Frame*;

**Returns**

This method returns a String object that contains the title of the Frame window as a text string.

**Example**

This example prints the title of the Frame window on the standard output device.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
        Frame f = new Frame("Drawing Tool");
        // extract and print the title of the Frame window
        System.out.println("The title on this frame is :" +
f.getTitle());
        f.resize(200, 200);
        f.show();
    }
}
```

## isResizable()

**ClassName**

Frame

**Purpose**

Indicates whether this Frame window object is resizable or not.

**Syntax**

public boolean isResizable()

**Parameters**

None.

**Description**

This method is used to test whether or not the dimensions of this Frame object can be changed.

**Imports**

*import java.awt.Frame;*

**Returns**

If the Frame object is resizable, then the return value is true; otherwise this method returns false.

**Example**

The following code illustrates the use of this function in determining whether a Frame object is resizable or not.

```
import java.awt.*;
public class FrameTest {
    ....
    ....
```

```
    // method that toggles the state of the resizable property of the
           specified Frame window
    public toggleFrame(Frame f) {
        if (f.setResizable());          // is the Frame resizable ?
            f.setResizable(true);   // make the Frame resizable
        else
            f.setResizable(false);  // disable resizable property
    }
}
```

## paramString()

**ClassName**

Frame

**Purpose**

To return the parameter values associated with this Frame object

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

The parameter values for this Frame object are returned as a String. This protected method cannot be invoked directly. The toString method of the Component superclass invokes this method.

**Imports**

*import java.awt.Frame*;

**Returns**

A String containing the parameter values for this Frame object.

**See Also**

The toString and paramString methods of the Component class

**Example**

The following example prints the parameter information for a Frame window object.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
        Frame f = new Frame("Testing Frame.paramString()"); //
application
            title
        System.out.println("f.toString() = " + f.toString());
        f.resize(200, 200);
        f.show();
    }
}
```

When this example is compiled and executed, the following string is printed on the screen.

```
f.toString() = java.awt.Frame[0,0,0x0,invalid,hidden,
layout=java.awt.BorderLayout,resizable,title=Testing
Frame.paramString()]
```

## remove(MenuComponent)

**ClassName**

Frame

**Purpose**

To remove the specified menu bar object from the Frame window.

**Syntax**

public synchronized void remove(MenuComponent *m*)

**Parameters**

***m***

The MenuComponent object to be removed from this Frame.

**Description**

This method removes the specified menu bar from this Frame window.

**Imports**

*import java.awt.Frame*;

**Returns**

None.

**See Also**

The MenuComponent and MenuBar classes

**Example**

This function removes the menu bar from the specified Frame window.

```
import java.awt.*;

class FrameTest {
    // construct the frame window and attach a menu bar to it
    ...
    ...
    // method that removes a given Frame's menubar
    public removeMenu(Frame f) {
        MenuBar mb = f.getMenuBar(); // get the MenuBar object
        f.remove(mb);                // remove the menu bar from the
Frame
            window
        }
}
```

## setCursor(int)

**ClassName**

Frame

**Purpose**

To set the cursor to display when the pointer is within this Frame window.

**Syntax**

public void setCursor(int cursorType)

**Parameters**

***cursorType***

An integer constant that indicates the type of cursor to display within this Frame window. The cursor types defined in this class are

| | |
|---|---|
| CROSSHAIR_CURSOR | Cursor image is a crosshair |
| *DEFAULT_CURSOR* | Default cursor image(arrow cursor) |
| *E_RESIZE_CURSOR* | Cursor image when the window is being resized to the right |
| *HAND_CURSOR* | The image for the cursor is a small hand |
| *MOVE_CURSOR* | The cursor image when the window is being moved |
| *N_RESIZE_CURSOR* | Cursor image when the window is being resized upwards |
| *NE_RESIZE_CURSOR* | Cursor image when the window is being resized by dragging its north-east corner |
| *NW_RESIZE_CURSOR* | Cursor image when the window is being resized by dragging its north-west corner |
| *S_RESIZE_CURSOR* | Cursor image when the window is being resized downwards |
| *SE_RESIZE_CURSOR* | Cursor image when the window is being resized by dragging its south-east corner |
| *SW_RESIZE_CURSOR* | Cursor image when the window is being resized by dragging its south-west corner |
| *TEXT_CURSOR* | Cursor image when the cursor is in an editable text window |
| *W_RESIZE_CURSOR* | Cursor image when the window is being resized to the left |
| *WAIT_CURSOR* | Hourglass cursor image |

**Description**

This method specifies the cursor image to display when the pointer is within this Frame window. The cursor can be any one of the types listed in the Parameters section of this method.

**Imports**

*import java.awt.Frame*;

**Returns**

None.

**Example**

This code causes the cursor image to change to an image of a hand when the mouse pointer is inside the Frame window,.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
        Frame f = new Frame("Hand Cursor");   // application title
        f.setCursor(Frame.HAND_CURSOR);       // set the cursor image
        f.resize(200, 200);
        f.show();
```

```
    }
}
```

## setIconImage(Image)

**ClassName**

Frame

**Purpose**

To set the image of the icon of this Frame object.

**Syntax**

public void setIconImage(Image *image*)

**Parameters**

***image***

The image to be used for the icon.

**Description**

This method specifies the image to use when the Frame is iconized. Some platforms do not support icons for windows.

**Imports**

*import java.awt.Frame;*

**Returns**

None.

**See Also**

The Image class

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
        Frame f = new Frame("Testing Frame.setIconImage()");
        // application title
        // create an Image object
        Image im = .......

        // assign it to the Frame as the Frame's icon image
        f.setIconImage(im);
        f.resize(200, 200);
        f.show();
    }
}
```

## setResizable(boolean)

**ClassName**

Frame

**Purpose**

To enable/disable the resizable feature on this Frame object.

**Syntax**

public void setResizable(boolean resizable)

**Parameters**

***resizable***

A boolean value that represents whether or not the Frame window is resizable. If set to true, the Frame object is resizable; if set to false, the Frame window is not resizable.

**Description**

This method is used to set whether the Frame window is resizable or not.

**Imports**

*import java.awt.Frame;*

**Returns**

None.

**Example**

The following function toggles the resizable flag of the specified Frame object.

```
import java.awt.*;

public class FrameTest {
    ....
    ....
    // method that toggles the state of the resizable property of the
        specified Frame window
    public toggleFrame(Frame f) {
        if (f.isResizable())
            f.setResizable(true);   // make the Frame resizable
        else
            f.setResizable(false);  // disable resizable property
    }
}
```

## setTitle(String)

**ClassName**

Frame

**Purpose**

Sets the text string on the title bar of the frame.

**Syntax**

public void setTitle(String *title*)

**Parameters**

***title***

The text string to display in the title bar of this Frame.

**Description**

This method sets the text label displayed in the title bar of the Frame window to the specified string.

**Imports**

*import java.awt.Frame;*

**Returns**

None.

**Example**

This method is used in the following example to set the title of a Frame window.

```
import java.awt.*;

public class FrameTest {
    public static void main(String args[]) {
        Frame f = new Frame();         // a Frame with no title
        // set the title of the Frame window
        f.setTitle("Countries and Capitals");
        f.resize(200, 200);
        f.show();
    }
}
```

## Insets

**Purpose**

Specifies an inset from within a rectangular area.

**Syntax**

public class Inset extends Object implements Cloneable

**Description**

This class represents the top, left, bottom, and right insets. This class is used to calculate the actual area that may be used inside a rectangular region, after subtracting the inset on each side of the region. This class is used by layout managers to lay out components. Figure 4-7 shows the inheritance hierarchy for the Insets class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.Insets;*

**Constructors**

public Insets(int *top*, int *left*, int *bottom*, int *right*)

**Parameters**

***top***

The distance set in from the top of the Container.

***left***

The distance set in from the left of the Container.

***bottom***

The distance set in from the bottom of the Container.

***right***

The distance set in from the right of the Container.

**Example**

This code shows how to construct an Insets object.

```
import java.awt.Insets;

public class InsetsTest {
    public static void main(String args[]) {
        // Insets constructor
        Insets i = new Insets(5, 10, 5, 10);   // top, left, bottom,
          right inset
    }
```

```
}
```

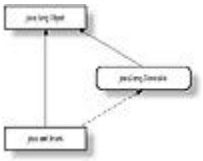

**Figure 4-7** Inheritance hierarchy for the Insets class

## clone()

**ClassName**
Insets
**Purpose**
To create a duplicate of this Insets object
**Syntax**
public Object clone()
**Parameters**
None.
**Description**
Creates a new instance of an Insets object and makes an exact duplicate of this Insets object.
**Imports**
*import java.awt.Insets;*
**Returns**
The return value is an Object that is a clone of this Insets object. You must cast this return value as an Insets object in order to use it as one.
**See Also**
The clone method of the Object class; the Cloneable interface
**Example**

```
import java.awt.Insets;

public class InsetsTest {
    public static void main(String args[]) {
        // Insets constructor
        Insets i1 = new Insets(5, 10, 5, 10);
        Insets i2 = (Insets)i1.clone();
        System.out.println("i1.toString() = " + i1.toString());
        System.out.println("i2.toString() = " + i2.toString());
    }
}
```

## toString()

**ClassName**
Insets
**Purpose**
To store the Insets object's parameter values in a String.
**Syntax**

public String toString()

**Parameters**

None.

**Description**

The values of the top, left, bottom, and right insets are returned as a String object.

**Imports**

*import java.awt.Insets;*

**Returns**

The return value is a String that contains the values of the parameters for the Insets object. The values are prefixed by a short textual description of the property they denote.

**See Also**

The Object class

**Example**

This method is implemented in the following example.

```
import java.awt.Insets;

public class InsetsTest {
    public static void main(String args[]) {
        // Insets constructor
        Insets i = new Insets(5, 10, 5, 10);  // top, left, bottom,
right
          inset
        System.out.println("i.toString() = " + i.toString());
    }
}
```

When this example is compiled and executed, the following string is printed on the screen

```
i.toString() = java.awt.Insets[top=5,left=10,bottom=5,right=10]
```

## Label

**Purpose**

A class that represents a single line text label.

**Syntax**

public class Label extends Component

**Description**

This class implements a graphical object that displays a single line of non-editable text. The alignment of the text can be specified. By default, label text is centered within the label. The methods of the Component class can be applied to this class. Figure 4-8 shows the inheritance hierarchy for the Label class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.Label;*

**Constructors**

public Label()
public Label(String *label*)
public Label(String *label*, int *alignment*)

**Parameters**

***label***

The text string to display in the label.

***alignment***

The alignment mode for the text string's position in the label.

**Example**

The following example demonstrates the construction of Label objects.

```
import java.awt.*;

public class LabelTest extends Frame {
    LabelTest() {
        super("Testing Label constructors");      // application title
        Label l = new Label("First label");
        add("North", l);
        add("Center", new Label("Second label"));
        add("South", new Label("Right aligned label", Label.RIGHT));
        resize(150, 100);
        show();
    }

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

**Figure 4-8** Inheritance hierarchy for the Label class

## addNotify()

**ClassName**

Label

**Purpose**

To create a peer object for this Label object.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

This method creates a peer for this Label object, which you can use to change the appearance of the label without changing its functionality. The addNotify method is the earliest stage in the creation of a component where platform specific resources (such as color, fonts, and fontmetrics) may be determined. Classes that override this method must first call super.addNotify() before doing any other processing in this method.

**Imports**

*import java.awt.Label*;

**Returns**

None.

**See Also**

The LabelPeer interface; the Component class

**Example**

Refer to the example listed in the addNotify method of the Canvas class in this chapter, and also to the section on LabelPeer interfaces in Chapter 9.

## getAlignment()

**ClassName**

Label

**Purpose**

To get the alignment mode of the text string displayed on the label.

**Syntax**

public int getAlignment()

**Parameters**

None.

**Description**

This method gets the current alignment of the text displayed on the label. The mode can be one of Label.RIGHT, Label.CENTER, or Label.LEFT.

**Imports**

*import java.awt.Label*;

**Returns**

This method returns an integer that specifies the alignment of the label.

**Example**

The following example uses this method to determine a label's alignment.

```
import java.awt.*;

public class LabelTest extends Frame {
    LabelTest() {
        super("Testing Label.getAlignment()");   // application title
        Label l = new Label("Plain label", Label.LEFT);
        add("Center", l);
        int align = l.getAlignment();
        switch (align) {
            case Label.LEFT:
            System.out.println("Label text is left aligned");
            break;

            case Label.CENTER:
```

```
            System.out.println("Label text is centered");
            break;

            case Label.RIGHT:
            System.out.println("Label text is right aligned");
            break;

        }

        resize(150, 100);
        show();
    }

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

## getText()

**ClassName**

Label

**Purpose**

To get the text string of this Label object.

**Syntax**

public String getText()

**Parameters**

None.

**Description**

This method gets the text string displayed on the label.

**Imports**

*import java.awt.Label*;

**Returns**

This method returns a String object that contains the text string displayed on the Label.

**See Also**

The setText method

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class LabelTest extends Frame {
    LabelTest()
        super("Testing Label.getText()");      // application title
        Label l = new Label("Temperature:");   // construct a label
        System.out.println(l.getText());       // extract and print
                                                  the label's text
        add("Center", l);
        resize(150, 100);
        show();
    }
```

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

## paramString()

**ClassName**

Label

**Purpose**

To return the parameter string associated with this Label object.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method returns the parameter values associated with the label (such as x, y coordinates, label text, and so on) as a String object. This protected method cannot be invoked from an application, but is invoked by the toString method of the Component class.

**Imports**

*import java.awt.Label*;

**Returns**

The return value is a String that contains the values of the parameters for the Label object. The values are prefixed by a short textual description of the property they denote.

**See Also**

The toString and paramString methods of the Component class

**Example**

This example prints the parameter values of a label.

```
import java.awt.*;

public class LabelTest extends Frame {
    LabelTest() {
        super("Testing Label.paramString()");     // application title
        Label l = new Label("Play", Label.LEFT);  // create a label
        System.out.println(l.toString());         // print parameter
                                                     information
        add("Center", l);
        resize(150, 100);
        show();
    }

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

## setAlignment(int)

**ClassName**

Label

**Purpose**

To set the alignment of the text string on the label.

**Syntax**

public void setAlignment(int alignment)

**Parameters**

***alignment***

The alignment mode to use for positoining the text on the label.

**Description**

This method sets the alignment mode to use for positioning the text on the label. If an invalid value is passed as an alignment mode, then an IllegalArgumentException is thrown.

**Imports**

*import java.awt.Label*;

**Returns**

None.

**See Also**

The getAlignment method of the IllegalArgumentException class

**Example**

In the following example this method is used to set the alignment mode of a label.

```
import java.awt.*;

public class LabelTest extends Frame {
    LabelTest() {
        super("Testing Label.setAlignment()");   // application title
        // create a label, the default alignment is Label.CENTER
        Label l = new Label("Record");
        l.setAlignment(Label.RIGHT);             // force it to be
                                                    left aligned
        add("Center", l);
        resize(150, 100);
        show();
    }

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

## setText(String)

**ClassName**

Label

**Purpose**

To change the text string displayed on the label.

**Syntax**

public void setText(String label)

**Parameters**

***label***

The text for the label.

**Description**

This method sets the text string displayed on the label.

**Imports**

*import java.awt.Label*;

**Returns**

None.

**See Also**

The getText method

**Example**

This method is used in the following sample code.

```
import java.awt.*;

public class LabelTest extends Frame {
    LabelTest() {
        super("Testing Label.setText()");   // application title
        Label l = new Label();              // construct an empty label
        l.setText("Month name:");           // now, put some text in it
        add("Center", l);
        resize(150, 100);
        show();
    }

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

## Panel

**Purpose**

A class that implements a generic container in which other components can be laid out.

**Syntax**

public class Panel extends Container

**Description**

Panels are commonly used as windows in which to arrange other components (such as Buttons, Labels, etc.). The FlowLayout layout manager is the default layout manager used for all Panels. Panels do not have a title bar and, unlike Frame windows, they cannot be used as top-level windows. The methods of the Container and Component classes can be invoked on Panels. Refer to the examples in Chapter 2 and Chapter 3 for more information. Figure 4-9 shows the inheritance hierarchy for the Panel class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.Panel*;

**Constructors**

public Panel()

**Parameters**

None.

**Example**

In this example, buttons are added to a Panel window within a Frame window.

```
import java.awt.*;

public class PanelTest {
    public static void main(String args[]) {
        Frame f = new Frame("Panel Test");  // application
                                               top-level window
        Panel p = new Panel();              // create a Panel
        p.setBackground(Color.yellow);
        p.add(new Label("Labels"));         // add components to
                                               the panel
        p.add(new Label("in"));
        p.add(new Label("a"));
        p.add(new Label("Panel"));
        f.add("South", p);                  // add the Panel to the
Frame
        f.resize(200, 200);
        f.show();
    }
}
```

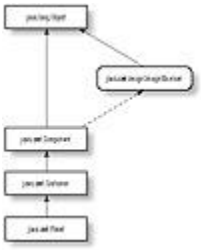

**Figure 4-9** Inheritance hierarchy for the Panel class

## addNotify()

**ClassName**

Panel

**Purpose**

To create a peer for this Panel object.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

This method creates a peer for this Panel object that you can use to change the appearance of the panel window without changing its functionality. The addNotify method is the earliest stage in the creation of a component at which

platform specific resources (such as color, fonts, and fontmetrics) may be determined. Classes that override this method must first call super.addNotify() before doing any other processing in this method.

**Imports**

*import java.awt.Panel*;

**Returns**

None.

**See Also**

The PanelPeer interface

**Example**

Refer to the example listed in the addNotify method of the Canvas class in this chapter and also to the section on PanelPeer interfaces in Chapter 9.

## Scrollbar

**Purpose**

A class that represents a scrollbar.

**Syntax**

public class Scrollbar extends Component

**Description**

This class implements a scrollbar object. Applications use scrollbars to scroll the data or image displayed on the screen. The scrollbar thumb position indicates the position of the visible portion of the image or data within a larger image or data buffer. Scrollbars are associated with a viewing area, and by dragging the thumb of the scrollbar, a user can change the image or data displayed in the viewing area. Figure 4-10 shows the inheritance hierarchy for the Scrollbar class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.Scrollbar*;

**Constructors**

public Scrollbar()
public Scrollbar(int orientation)
public Scrollbar(int orientation, int value, int visible, int minimum, int maximum)

**Parameters**

***orientation***

The orientation of the scrollbar.

***value***

The current value of the scrollbar's thumb position.

***visible***

The size of the visible region of the area that is being scrolled using the scrollbar.

***minimum***

The minimum value of the scrollbar.

***maximum***

The maximum value of the scrollbar.

**Example**

This sample code shows how to construct Scrollbar objects.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);                  // application title
        resize(200, 300);
        Rectangle r = bounds();        // determine dimensions of
                                          this Frame window
        Scrollbar sb1 = new Scrollbar();   // default orientation
                                              is vertical
        add("East", sb1);              // attach it to the right of
                                          the frame
//      Scrollbar sb2 = new Scrollbar(Scrollbar.HORIZONTAL);
//      add("South", sb2);
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
         0, r.width);
        add("South, sb3);              // attach this scrollbar to
                                          the bottom
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing scrollbars");
    }
}
```

**Figure 4-10** Inheritance hierarchy for the Scrollbar class

## addNotify()

**ClassName**

ScrollBar

**Purpose**

Creates a peer object for this Scrollbar object.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

Using this method one can change the appearance of the scrollbar without changing its functionality. The addNotify method is the earliest stage in the creation of a component at which platform specific resources such as color, fonts

and fontmetrics may be determined. Classes that override this method *must* first call super.addNotify() before doing any other processing in this method.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

None.

**See Also**

The ScrollbarPeer class

**Example**

Refer to the example listed in the addNotify method of the Canvas class in this chapter and also to the section on ScrollbarPeer interfaces in Chapter 9.

## getLineIncrement()

**ClassName**

Scrollbar

**Purpose**

To get the step size, set for decrements/increments when the line up/down arrow buttons of the scrollbar are invoked.

**Syntax**

public int getLineIncrement()

**Parameters**

None.

**Description**

This method returns an integer that represents the step size that will increment line.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

This method returns an integer that represents the line increment step size.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);           // application title
        resize(200, 300);
        Rectangle r = bounds(); // determine dimensions of this
                                   Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3)       // attach this scrollbar to the bottom
        System.out.println("sb3. getLineIncrement = " + sb3.
          getLineIncrement());
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.getLineIncrement()");
```

```
    }
}
```

## getMaximum()

**ClassName**

Scrollbar

**Purpose**

To determine the value of the maximum position of the scrollbar thumb

**Syntax**

public int getMaximum**()**

**Parameters**

None.

**Description**

This method gets the value for the maximum position of the thumb for this Scrollbar object.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

This method returns an integer value that represents the maximum position of the Scrollbar object.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);                 // application title
        resize(200, 300);
        Rectangle r = bounds();       // determine dimensions of
                                         this Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);            // attach this scrollbar to
                                         the bottom
        System.out.println("sb3. getMaximum = " +
          sb3.getMaximum());
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.getMaximum()");
    }
}
```

## getMinimum()

**ClassName**

Scrollbar

**Purpose**

To determine the minimum position of the scrollbar thumb.

**Syntax**

public int getMinimum()

**Parameters**

None.

**Description**

This method gets the value for the minimum position of the thumb for this Scrollbar object.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

This method returns an integer that represents the minimum position of the Scrollbar object.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);              // application title
        resize(200, 300);
        Rectangle r = bounds();    // determine dimensions of this
                                      Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);         // attach this scrollbar to the
bottom
        System.out.println("sb3.getMinimum = " +
          sb3.getMinimum());
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.getMinimum()");
    }
}
```

## getPageIncrement()

**ClassName**

Scrollbar

**Purpose**

Gets the step size, set for decrements/increments when the page up/down actions of the scrollbar are invoked.

**Syntax**

public int getPageIncrement()

**Parameters**

None.

**Description**

This method returns an integer that represents the page increment step size.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

This method returns an integer that represents the page increment step size.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);            // application title
        resize(200, 300);
        Rectangle r = bounds();  // determine dimensions of this
                                    Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);       // attach this scrollbar to the bottom
        System.out.println("sb3.getPageIncrement = " +
          sb3.getPageIncrement());
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.getPageIncrement()");
    }
}
```

## getOrientation()

**ClassName**

Scrollbar

**Purpose**

To determine the orientation of the scrollbar.

**Syntax**

public int getOrientation()

**Parameters**

None.

**Description**

This method gets the value for the orientation of this Scrollbar object.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

This method returns an integer that represents the orientation of this ScrollBar object. The value returned is either Scrollbar.HORIZONTAL or Scrollbar.VERTICAL.

**Example**

The following example demonstrates the use of this method in an application.

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

```
public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);                       // application title
        resize(200, 300);
        Scrollbar sb1 = new Scrollbar();    // default orientation
                                               is vertical
        add("East", sb1);                   // attach it to the
                                               right of the frame
        if (sb1.getOrientation() == Scrollbar.VERTICAL)
            System.out.println("Vertical scrollbar");
        else
            System.out.println("Horizontal scrollbar");
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing scrollbars");
    }
}
```

## getValue()

**ClassName**

Scrollbar

**Purpose**

Determines the value of the current position of the scrollbar thumb

**Syntax**

public int getValue()

**Parameters**

None.

**Description**

This method gets the value for the current position of the thumb for this Scrollbar object.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

This method returns an integer that represents the current position of the Scrollbar object.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);                 // application title
        resize(200, 300);
        Rectangle r = bounds();       // determine dimensions of
                                         this Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);            // attach this scrollbar to
                                         the bottom
```

```
        System.out.println("sb3.getValue = " + sb3.getValue());
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing getValue");
    }
}
```

## getVisible()

**ClassName**

Scrollbar

**Purpose**

To determine the size of the visible portion of the scrollbar.

**Syntax**

public int getVisible()

**Parameters**

None.

**Description**

This method gets the value for the visible portion of this Scrollbar object.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

This method returns an integer that represents the visible portion of this Scrollbar object.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);              // application title
        resize(200, 300);
        Rectangle r = bounds();    // determine dimensions of this
                                      Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);         // attach this scrollbar to the
bottom
        System.out.println("sb3.getVisible = " +
          sb3.getVisible());
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
           Scrollbar.getVisible()");
    }
}
```

## paramString()

**ClassName**

Scrollbar

**Purpose**

To return the parameter string associated with this Scrollbar object

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method returns the parameter values associated with a Scrollbar object. The values are prefixed by short descriptive tags. This protected method cannot be invoked from an application, but is invoked by the toString method of the Component class.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

The return value is a String that contains the values of the parameters for the Scrollbar object. The values are prefixed by a short textual description of the property they denote.

**See Also**

The toString and paramString methods of the Component class

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);                // application title
        resize(200, 300);
        Rectangle r = bounds();      // determine dimensions of
                                        this Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);           // attach this scrollbar to
                                        the bottom
        System.out.println("sb3.toString = " + sb3.toString());
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.toString()");
    }
}
```

## setLineIncrement(int)

**ClassName**

Scrollbar

**Purpose**

Sets the step size for decrements/increments when the line up/down arrow buttons of the scrollbar are invoked.

**Syntax**

public void setLineIncrement(int l)

**Parameters**

***l***

The line increment size.

**Description**

This method specifies the amount that the area is to be scrolled when the user invokes the line up/down arrow buttons of the scrollbar. The position of the scrollbar thumb within the scrollbar is also updated proportional to the value specified in this method.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

None.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);            // application title
        resize(200, 300);
        Rectangle r = bounds();  // determine dimensions of this
                                    Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);       // attach this scrollbar to the bottom
        sb3.setLineIncrement(5);
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.setLineIncrement()");
    }
}
```

## setPageIncrement(int)

**ClassName**

Scrollbar

**Purpose**

Sets the step size for decrements/increments when the page up/down actions of the scrollbar are invoked.

**Syntax**

public void setPageIncrement(int l)

**Parameters**

***l***

The page increment size.

**Description**

This method specifies the amount that the area is to be scrolled when the user invokes the page up/down actions. The position of the scrollbar thumb within the scrollbar is also updated proportional to the value specified in this method

**Imports**

*import java.awt.Scrollbar;*

**Returns**

None.

**Example**

The following example demonstrates the use of this method in an application.

```
import java.awt.*;

public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);              // application title
        resize(200, 300);
        Rectangle r = bounds();    // determine dimensions of this
                                      Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);         // attach this scrollbar to the
bottom
        sb3.setPageIncrement(25);  // override the existing value
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.setPageIncrement()");
    }
}
```

## setValue(int)

**ClassName**

Scrollbar

**Purpose**

Sets the value of the current position of this Scrollbar to the specified value.

**Syntax**

public void setValue(int value)

**Parameters**

***value***

The new value for the current position of the Scrollbar. If this value is less than the minimum value of the scrollbar, then it becomes the new minimum value of the scrollbar. Similarly, if this value is more than the maximum value of the scrollbar, then it becomes the new maximum value of the scrollbar.

**Description**

This method sets the value for the current position of the thumb for this Scrollbar object.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

None.

**Example**

The following example demonstrates the use of this method in an application.

```
public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title)              // application title
        resize(200, 300);
        Rectangel r = bounds();   // determine dimensions of this
                                     Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10,
          0, r.width);
        add("South", sb3);        // attach this scrollbar to the
bottom
        sb3.setValue(100);        // set the thumb in the middle
                                     of the scrollbar
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.setValue()");
    }
}
```

## setValues(int, int, int, int)

**ClassName**

Scrollbar

**Purpose**

Sets various parameters associated with this ScrollBar object.

**Syntax**

public void setValues(int value, int visible, int minimum, int maximum)

**Parameters**

***value***

The position of the scrollbar thumb in the current window.

***visible***

The size of the visible region of the area being scrolled using the scrollbar.

***minimum***

The minimum value of the scrollbar.

***maximum***

The maximum value of the scrollbar.

**Description**

This method provides a convenient way to set the various parameters of this Scrollbar object.

**Imports**

*import java.awt.Scrollbar;*

**Returns**

None.

**Example**

The following example demonstrates the use of this method in an application.

```
public class TestScroll extends Frame {
    TestScroll(String title) {
        super(title);             // application title
        resize(200, 300);
        Rectangel r = bounds();   // determine dimensions of this
                                     Frame window
        Scrollbar sb3 = new Scrollbar(Scrollbar.HORIZONTAL);
        add("South", sb3);        // attach this scrollbar to the
                                     bottom
        sb3.setValues(0, 25, 0, r.width);   // value, visible, min, max
        show();
    }
    public static void main(String args[]) {
        TestScroll ts = new TestScroll("Testing
          Scrollbar.setValue()");
    }
}
```

## LayoutManager

**Purpose**

An interface for classes that need to lay out Components in Containers.

**Syntax**

public interface LayoutManager extends Object

**Description**

This interface is used to implement the mechanisms required of classes that know how to lay out Containers. All the layout managers supplied with the Java AWT implement this interface.

**PackageName**

*java.awt*

**Imports**

*import java.awt.LayoutManager*;

**Constructors**

None.

**Parameters**

None.

**Example**

See the examples for FlowLayout and GridLayout.

## addLayoutComponent(String, Component)

**Interface**

LayoutManager

**Purpose**

To add the specified component to the layout, associating the component with the specified name.

**Syntax**

public abstract void addLayoutComponent(String name, Component comp)

**Parameters**

***name***

Name of the area where the component should be added.

***comp***

Component object to be added.

**Description**

This method should be defined in any class that implements the LayoutManager interface.

**Imports**

*import java.awt.LayoutManager*;

**Returns**

None.

**See Also**

The Container class

**Example**

The following example demonstrates the implementation of this method.

```
import java.awt.*

public class MyLayout implements LayoutManager {
    /* Implement constructors for this class */

    public void addLayoutComponent(String name, Component comp) {
    /* Code to add the component to a specific named area
    of the layout goes here */
    }

    /* Implement all the other methods of the LayoutManager
       interface */
}
```

## layoutContainer(Container)

**Interface**

LayoutManager

**Purpose**

To lay out the specified container.

**Syntax**

public abstract void layoutContainer(Container parent)

**Parameters**

***parent***

The Container object to be laid out.

**Description**

This method causes the specified container to be laid out. It should be defined in any class that implements the LayoutManager interface.

**Imports**

*import java.awt.LayoutManager*;

**Returns**

None.

**See Also**

The Container class

**Example**

The following sample code for a custom layout manager shows the implementation of this function.

```
import java.awt.*

public class MyLayout implements LayoutManager {
    /* Implement constructors for this class */

    public void layoutContainer(Container parent){
        /* Calculate every component's size and position and lay
        out the components in the manner desired */
    }

    /* Implement all the other methods of the LayoutManager
       interface */
}
```

## minimumLayoutSize(Container)

**Interface**

LayoutManager

**Purpose**

To calculate the minimum size required to lay out the container, taking into account the components in the specified container.

**Syntax**

public abstract Dimension minimumLayoutSize(Container parent)

**Parameters**

***parent***

The Container that holds the components that need to be laid out.

**Description**

This method should be defined in any class that implements the LayoutManager interface.

**Imports**

*import java.awt.LayoutManager*;

**Returns**

The return type of this method is Dimension. This return value contains the minimum height and width required to layout the container in the specified panel.

**See Also**

The Container class

**Example**

Here is an extract from a class that implements the LayoutManager interface.

```
import java.awt.*
```

```
public class MyLayout implements LayoutManager {
    /* Implement constructors for this class */

    public Dimension minimumLayoutSize(Container parent) {
        Dimension dim = new Dimension(0, 0);

        /* Code to calculate the minimum width and height */

        return dim;
    }
    /* Implement all the other methods of the LayoutManager
       interface */
}
```

## preferredLayoutSize(Container)

**Interface**

LayoutManager

**Purpose**

To calculate the preferred dimensions required to lay out the container, taking into account the components in the specified container.

**Syntax**

public abstract Dimension preferredLayoutSize(Container parent)

**Parameters**

***parent***

The Container that holds the components that need to be laid out.

**Description**

This method should be defined in any class that implements the LayoutManager interface.

**Imports**

*import java.awt.LayoutManager*;

**Returns**

The return type of this method is Dimension. This return value contains the preferred height and width required to lay out the container in the specified panel.

**See Also**

The Container Class

**Example**

The following example demonstrates the implementation of this method.

```
import java.awt.*

public class MyLayout implements LayoutManager {
    /* Implement constructors for this class */

    public Dimension preferredLayoutSize(Container parent) {
    Dimension dim = new Dimension(0, 0);

    /* Code to calculate the ideal width and height */
    return dim;
    }
    /* Implement all the other methods of the LayoutManager
```

```
       interface */
}
```

## removeLayoutComponent(Component)

**Interface**

LayoutManager

**Purpose**

To remove the specified component from the layout.

**Syntax**

public abstract void removeLayoutComponent(Component comp)

**Parameters**

***comp***

The Component to be removed from the layout.

**Description**

This method should be defined in any class that implements the LayoutManager interface.

**Imports**

*import java.awt.LayoutManager*;

**Returns**

None.

**See Also**

The Container class

**Example**

The following sample code demonstrates the implementation of this method in a class that implements a custom layout manager.

```
import java.awt.*

public class MyLayout implements LayoutManager {
    /* Implement constructors for this class */

    public void removeLayoutComponent(Component comp) {
    /* Code to remove the specified component goes here */
    }

    /* Implement all the other methods of the LayoutManager
       interface */
}
```

## FlowLayout

**Purpose**

A simple layout manager that lays out components in rows (from left to right).

**Syntax**

public class FlowLayout extends Object implements LayoutManager

**Description**

This class implements the LayoutManager interface and is used to lay out components in rows. The components are laid out from left to right and centered

within their row. This is the default layout manager for all Panels. Figure 4-11 shows the inheritance hierarchy for the FlowLayout class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.flowlayout;*

**Constructorts**

public FlowLayout()
public FlowLayout(int *align*)
public FlowLayout(int *align*, int *hgap*, int *vgap*)

**Parameters**

***align***

The alignment to use for laying out components (can be LEFT,CENTER, or RIGHT).

***hgap***

The horizontal gap to leave between components.

***vgap***

The vertical gap to leave between components.

**Example**

This sample code shows how to construct FlowLayout objects using the different FlowLayout constructors.

```
/* Default FlowLayout constructor */
FlowLayout f1 = new FlowLayout();

/* FlowLayout object that aligns components to the left */
FlowLayout f2 = new FlowLayout(FlowLayout.LEFT);

/* FlowLayout object that aligns components to the left, with a
   horizontal gap of 30 units between components and a vertical gap
   of 0 units */
FlowLayout f3 = new FlowLayout(FlowLayout.LEFT, 30, 0);
```

**Figure 4-11** Inheritance hierarchy for the FlowLayout class

## addLayoutComponent(String, Component)

**ClassName**

FlowLayout

**Purpose**

To add the specified component to the area in the layout associated with the specified name.

**Syntax**

public void addLayoutComponent(String name, Component comp)

**Parameters**

***name***

Name of the component to be added.

***comp***

Component object to be added.

**Description**

The FlowLayout class does not divide the layout area into subareas, and hence does not need to implement any functionality for this method. In order to conform to the LayoutManager interface, this method is an empty stub in the FlowLayout implementation.

**Imports**

*import java.awt.FlowLayout;*

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

This code invokes the Container's add method, which in turn invokes this method.

```
/* Create a Container for the components */
Panel p = new Panel();
/* set the layout manager for this panel */
p.setLayout(new FlowLayout());

/* add the components to be laid out */
/* This results in the component being added to the layout manager */
p.add("North", new Button("This"));          // the named areas carry
                                          significance only for the
p.add("West", new Button("demonstration"));  // BorderLayout layout
                                                manager, other layout
p.add("Center",  new Button("is"));          // managers ignore
                                                this parameter
p.add("East",  new Button("really"));
p.add("South", new Button("cool !"));
```

## layoutContainer(Container)

**ClassName**

FlowLayout

**Purpose**

To lay out the specified container in rows, aligning the components within each row.

**Syntax**

public void layoutContainer(Container parent)

**Parameters**

***parent***

Container object to be laid out.

**Description**

The components are laid out from left to right and aligned within a row. The default alignment is CENTER, but a different alignment (either LEFT or RIGHT) can be specified while constructing the FlowLayout object. The horizontal gap between components in a row and the vertical gap between rows can also be specified in the FlowLayout constructor. Applications do not directly invoke this method. The layout method of the Container class results in a call to this method. The layout method of the Container class is invoked when the Container needs to be displayed on the screen.

**Imports**

*import java.awt.FlowLayout;*

**Returns**

None.

**See Also**

The LayoutManager interface; the Container class

**Example**

In the following example, a Frame window is created and displayed.

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

public class LayoutDemo extends Applet {
    /* Main method to start running the applet */
    public static void main(String args[]) {
        Frame f = new Frame("Layout Demonstration");
        f.setLayout(new FlowLayout());
        Button b = new Button("Testbutton");
        f.add("Center", b);
        f.resize(500, 200);
        /* this causes the layout manager associated with
           this Container to lay out the Container and its
             Components on the screen */
        f.show();
    }
}
```

## minimumLayoutSize(Container)

**ClassName**

FlowLayout

**Purpose**

To calculate the minimum size required to lay out the container, taking into account the components in the specified container.

**Syntax**

public Dimension minimumLayoutSize(Container parent)

**Parameters**

***parent***

Container containing components that need to be laid out.

**Description**

This method calculates and returns the minimum dimensions required by the FlowLayout manager to lay out the components.

**Imports**

*import java.awt.FlowLayout;*

**Returns**

The return type of this method is Dimension. This return value contains the minimum height and width required to lay out the container in the specified panel.

**See Also**

The Container class; the LayoutManager interface

**Example**

The following sample code is an example of this method in an application.

```
/* Create a Container for the components */
Panel p = new Panel();
/* set the layout manager for this panel */
p.setLayout(new FlowLayout());

/* add the components to the Container */
/* This results in the component being added to the layout manager */
p.add("North",  new Button("This"));
p.add("Center",  new Button("is"));
p.add("South", new Button("cool !"));

/* Get the minimum dimensions of the Container */
/* This results in a call to the minimumLayoutSize() method of the
   layout manager */
Dimension d = p.minimumSize();
```

## preferredLayoutSize(Container)

**ClassName**

FlowLayout

**Purpose**

To calculate the preferred dimensions for this layout, taking into account the components in the specified container.

**Syntax**

public Dimension preferredLayoutSize(Container target)

**Parameters**

***target***

Container that needs to be laid out.

**Description**

This method computes the ideal width and height required by this layout. The values returned have no effect unless the program specifically enforces these dimensions.

**Imports**

*import java.awt.FlowLayout;*

**Returns**

The return type of this method is Dimension. This return value contains the ideal height and width required to lay out the container in the specified panel.

**See Also**

The Container class; the LayoutManager interface

**Example**

The following sample code demonstrates the use of this method.

```
/* Create a Container for the components */
Panel p = new Panel();
/* set the layout manager for this panel */
p.setLayout(new FlowLayout());

/* add the components to the Container */
p.add("North",  new Button("This"));
p.add("Center",  new Button("is"));
p.add("South", new Button("cool !"));

/* Get the ideal dimensions of the Container */
/* This results in a call to the preferredLayoutSize()
   method of the layout manager */
Dimension d = p.preferredSize();
```

## removeLayoutComponent(Component)

**ClassName**

FlowLayout

**Purpose**

To remove the specified component from the layout.

**Syntax**

public void removeLayoutComponent(Component comp)

**Parameters**

***comp***

Component to be removed from the layout.

**Description**

This method is a dummy stub in the FlowLayout class, as this layout manager doesn't need to maintain associations between components and areas on the display. Applications do not directly invoke this method. The remove method of the Container class results in a call to this method.

**Imports**

*import java.awt.FlowLayout;*

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

The sample code adds buttons to a Panel and then removes one of the buttons.

```
/* Create a Container for the components */
Panel p = new Panel();
/* set the layout manager for this panel */
p.setLayout(new FlowLayout());

/* add the components to be laid out */
/* This results in the component being added to the layout manager */
Button b1 = new Button("Good");
```

```
Button b2 = new Button("Bad");
Button b3 = new Button("Day");
/* add the buttons to the container */
p.add(b1);
p.add(b2);
p.add(b3);
/* remove a specific button from the container */
/* The Container's remove() method invokes the layout
   manager's removeComponent() method */
p.remove(b2);
```

## toString()

**ClassName**

FlowLayout

**Purpose**

To represent the FlowLayout object's values as a String.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

The values of the horizontal gap variable, the vertical gap variable, and the alignment mode variable for this layout are returned as a String object.

**Imports**

*import java.awt.FlowLayout;*

**Returns**

The return value is a String that contains the values of the properties for the FlowLayout object. The values are prefixed by a short textual description of the property they denote.

**See Also**

The toString method of the Object class

**Example**

This method is implemented in the following function.

```
void printLayoutInfo(LayoutManager layout) {
        /* print the parameter values of the specified layout
           manager */
        System.out.println("layout.toString(): " +
           layout.toString());
}
```

## GridLayout

**Purpose**

A layout manager that creates a grid with the specified number of rows and columns and lays out components on the grid.

**Syntax**

public class GridLayout extends Object implements LayoutManager

**Description**

This class implements the LayoutManager interface and is used to lay out components in grids. It makes all the components of equal size and lays them out on the grid. Figure 4-12 shows the inheritance hierarchy for the GridLayout class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.GridLayout*;

**Constructors**

public GridLayout(int rows, int cols)
public GridLayout(int rows, int cols, int hgap, int vgap)

**Parameters**

***rows***

The number of rows in the grid.

***cols***

The number of columns in the grid.

***hgap***

The horizontal gap to leave between components.

***vgap***

The vertical gap to leave between components.

**Example**

Here is sample source code that illustrates GridLayout construction.

```
/* GridLayout constructor to lay out components in a single row
   with any number of columns */
GridLayout g1 = new GridLayout(1, 0);

/* GridLayout constructor specifying a grid of 3 rows and 2 columns */
GridLayout g2 = new GridLayout(3, 2);

/* GridLayout object that lays out components in a grid consisting
   of 3 rows and 2 columns. The horizontal gap between columns is
   20 units and the vertical gap between rows is 10 units */
  GridLayout g3 = new GridLayout(3, 2, 20, 10);
```

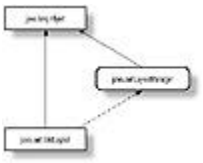

**Figure 4-12** Inheritance hierarchy for the GridLayout class

## addLayoutComponent(String, Component)

**ClassName**

GridLayout

**Purpose**

To add the specified component to the layout, associating the component with the specified name.

**Syntax**

public void addLayoutComponent(String name, Component comp)

**Parameters**

***name***

Name of the component to be added.

***comp***

Component object to be added.

**Description**

The GridLayout class does not divide the layout area into subareas, and hence does not need to implement any functionality for this method. In order to conform to the LayoutManager interface, this method is an empty stub in the GridLayout implementation.

**Imports**

*import java.awt.GridLayout*;

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## layoutContainer(Container)

**ClassName**

GridLayout

**Purpose**

To lay out the specified container in rows, aligning the components within each row.

**Syntax**

public void layoutContainer(Container parent)

**Parameters**

***parent***

Container object to be laid out.

**Description**

This method lays out the components in the container, in a grid. Applications do not directly invoke this method. The layout method of the Container class results in a call to this method.

**Imports**

*import java.awt.GridLayout*;

**Returns**

None.

**See Also**

The LayoutManager interface; the Container class

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## minimumLayoutSize(Container)

**ClassName**

GridLayout

**Purpose**

To calculate the minimum size required to lay out the container, taking into account the components in the specified container.

**Syntax**

public Dimension minimumLayoutSize(Container parent)

**Parameters**

***parent***

Container that needs to be laid out.

**Description**

This method calculates and returns the minimum dimensions required by the GridLayout manager to lay out the components contained within the specified container. The minimum dimensions are calculated according to the following formulae:

Minimum width = (Left + Right insets of *parent*) + (number of columns * width of widest component in *parent*) + ((number of columns -1)*inter-column gap)

Minimum height = (Top + Bottom insets of *parent*) + (number of rows * height of tallest component in *parent*) + ((number of rows -1)*inter-row gap)

**Imports**

*import java.awt.GridLayout;*

**Returns**

The return type of this method is Dimension. This return value contains the minimum height and width required to lay out the container in the specified panel.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## preferredLayoutSize(Container)

**ClassName**

GridLayout

**Purpose**

To calculate the preferred dimensions for this layout, taking into account the components in the specified container.

**Syntax**

public Dimension preferredLayoutSize(Container target)

**Parameters**

***target***

Container that needs to be laid out.

**Description**

This method computes the ideal width and height required by this layout. The values returned have no effect unless the program specifically enforces these dimensions.

**Imports**

*import java.awt.GridLayout*;

**Returns**

This method returns a Dimension object. This return value contains the ideal height and width required to lay out the container in the specified panel.

**See Also**

The container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## removeLayoutComponent(Component)

**ClassName**

GridLayout

**Purpose**

To remove the specified component from the layout.

**Syntax**

public void removeLayoutComponent(Component comp)

**Parameters**

***comp***

Component to be removed from the layout.

**Description**

This method is a dummy stub in the GridLayout class, as this layout manager doesn't need to maintain associations between components and areas on the display.

**Imports**

*import java.awt.GridLayout*;

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## toString()

**ClassName**

GridLayout

**Purpose**

To represent the values of the GridLayout object as a String.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method returns a String representation of this object's values, namely, the horizontal gap, the vertical gap, the number of rows, and the number of columns. Each value is prefixed with a short descriptive tag.

**Imports**

*import java.awt.GridLayout*;

**Returns**

This method returns a String containing the values of the GridLayout object, with each value prefixed by a descriptive tag.

**See Also**

The toString method of the Object class

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## BorderLayout

**Purpose**

A layout manager that divides a rectangular area into five named areas and lays out components in each of these named areas.

**Syntax**

public class BorderLayout extends Object implements LayoutManager

**Description**

This layout manager divides the area of the Container into five named areas: North, South, East, West, and Center. A container that uses this layout manager must add a component to a named area. The preferred dimensions of the components, added to the North, South, East, and West areas, are honored and the component added to the Center area occupies all the remaining space. This is the default layout manager for all Window objects (such as Frame windows and Dialog windows). Figure 4-13 shows the inheritance hierarchy for the BorderLayout class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.BorderLayout;*

**Constructors**

public BorderLayout()

public BorderLayout(int *hgap*, int

**Parameters**

***hgap***

The horizontal gap to leave between the named areas.

***vgap***

The vertical gap to leave between the named areas.

**Example**

Here is sample source code that illustrates BorderLayout construction.

```
/* default BorderLayout constructor */
BorderLayout b1 = new BorderLayout();

// horizontal gap between named areas is 20 units and the vertical
   gap is 10 units */
BorderLayout b2 = new BorderLayout(20, 10);
```

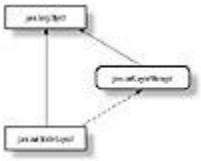

**Figure 4-13** Inheritance hierarchy for the BorderLayout class

## addLayoutComponent(String, Component)

**ClassName**

BorderLayout

**Purpose**

To add the component to the named area of the container.

**Syntax**

public void addLayoutComponent(String name, Component comp)

**Parameters**

***name***

Name of the area within the container to add the component to.

***comp***

Component object to be added.

**Description**

The BorderLayout layout manager divides the Container into five areas and hence, the parameter *name* can be one of North, South, East, West, or Center. The specified component is associated with the area and is added to that portion of the Container.

**Imports**

*import java.awt.BorderLayout;*

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## layoutContainer(Container)

**ClassName**

BorderLayout

**Purpose**

To lay out the specified container by laying out the components in the areas where they have been added.

**Syntax**

public void layoutContainer(Container parent)

**Parameters**

***parent***

Container object to be laid out.

**Description**

This method lays out the components in the container, in the named areas where they were added. Applications do not directly invoke this method. The layout method of the Container class results in a call to this method.

**Imports**

*import java.awt.BorderLayout;*

**Returns**

None.

**See Also**

The LayoutManager interface; the class Container

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## minimumLayoutSize(Container)

**ClassName**

BorderLayout

**Purpose**

To calculate the minimum size required to lay out the container, taking into account the components in the specified container.

**Syntax**

public Dimension minimumLayoutSize(Container parent)

**Parameters**

***parent***

Container that needs to be laid out.

**Description**

This method calculates and returns the minimum dimensions required by the BorderLayout manager to lay out the components contained within the specified container.

**Imports**

*import java.awt.BorderLayout;*

**Returns**

This method returns a Dimension object. This return value contains the minimum height and width required to lay out the container in the specified panel.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## preferredLayoutSize(Container)

**ClassName**

BorderLayout

**Purpose**

To calculate the preferred dimensions for this layout, taking into account the components in the specified container.

**Syntax**

public Dimension preferredLayoutSize(Container target)

**Parameters**

***target***

The Container that needs to be laid out.

**Description**

This method computes the ideal width and height required by this layout. The values returned have no effect unless the program specifically enforces these dimensions.

**Imports**

*import java.awt.BorderLayout;*

**Returns**

This method reutrns a Dimension object. This return value contains the ideal height and width required to lay out the container in the specified panel.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## removeLayoutComponent(Component)

**ClassName**

BorderLayout

**Purpose**

To remove the specified component from the layout.

**Syntax**

public void removeLayoutComponent(Component comp)

**Parameters**

***comp***

Component to be removed from the layout.

**Description**

This method disassociates the component being removed from the named area to which it was added.

**Imports**

*import java.awt.BorderLayout;*

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## toString()

**ClassName**

BorderLayout

**Purpose**

To represent the values of the BorderLayout object as a String.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method returns a String representation of this BorderLayout object's values, namely the horizontal gap and the vertical gap between the named areas, that the BorderLayout organizes its components in. Each value is prefixed with a short descriptive tag.

**Imports**

*import java.awt.BorderLayout;*

**Returns**

This method returns a String containing the values of the BorderLayout object, each value prefixed by a descriptive tag.

**See Also**

The toString method of the Object class

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## CardLayout

**Purpose**

A powerful layout manager that a container can use to lay out its components as a stack of cards, only one card being visible at any point of time.

**Syntax**

public class CardLayout extends Object implements LayoutManager

**Description**

This layout manager allows the Container to use the same real estate on the screen to present different views of different components. These views are arranged in a

fashion similar to that of a deck of cards and the individual views can be flipped back and forth on the view stack. Figure 4-14 shows the inheritance hierarchy for the CardLayout class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.CardLayout;*

**Constructors**

public CardLayout()
public CardLayout(int hgap, int vgap)

**Parameters**

***hgap***

The horizontal gap to leave between components on each card.

***vgap***

The vertical gap to leave between components on each card.

**Example**

Here is sample source code that illustrates CardLayout construction. The chapter project demonstrates the use of this layout manager in a more complete manner.

```
CardLayout c1 = new CardLayout();       // default constructor

// horizontal gap between named areas is 20 units and the vertical
   gap is 10 units */
CardLayout b2 = new CardLayout(20, 10);
```

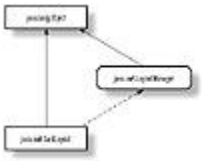

**Figure 4-14** Inheritance hierarchy for the CardLayout class

## addLayoutComponent(String, Component)

**ClassName**

CardLayout

**Purpose**

To add the specified component to the layout, associating the component with the specified name.

**Syntax**

public void addLayoutComponent(String name, Component comp)

**Parameters**

***name***

Name of the component to be added.

***comp***

Component object to be added.

**Description**

The CardLayout class maintains the stack of cards to display. This stack is updated as components are added to the card stack. The parameter *name* can be any user-specified string. This parameter name is used by the show method to bring the named card into view.

**Imports**

*import java.awt.CardLayout;*

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface; the show method of the CardLayout class

**Example**

The chapter project uses a CardLayout to display different views. Refer to it for more details.

## first(Container)

**ClassName**

CardLayout

**Purpose**

To make the first card in the card stack visible.

**Syntax**

public void first(Container parent)

**Parameters**

***parent***

The parent Container object that this CardLayout object is the layout manager for.

**Description**

The first card in the card stack is brought into view and all the Components on this card are displayed.

**Imports**

*import java.awt.CardLayout;*

**Returns**

None.

**Example**

This method is used in the project at the end of this chapter.

## last(Container)

**ClassName**

CardLayout

**Purpose**

To make the last card in the card stack visible.

**Syntax**

public void last(Container parent)

**Parameters**

***parent***

The parent Container object that this CardLayout object is the layout manager for.

**Description**

The card at the bottom of the card stack is brought into view.

**Imports**

*import java.awt.CardLayout;*

**Returns**

None.

**Example**

The project at the end of this chapter uses this method.

## layoutContainer(Container)

**ClassName**

CardLayout

**Purpose**

To lay out the specified container in rows, aligning the components within each row.

**Syntax**

public void layoutContainer(Container parent)

**Parameters**

***parent***

Container object to be laid out.

**Description**

This method lays out the components in the container, in the named areas where they were added. Applications do not directly invoke this method. The layout method of the Container class results in a call to this method.

**Imports**

*import java.awt.CardLayout;*

**Returns**

None.

**See Also**

The LayoutManager interface; the Container class

**Example**

Refer to the example given under the corresponding function in the FlowLayout class and to the chapter project.

## minimumLayoutSize(Container)

**ClassName**

CardLayout

**Purpose**

To calculate the minimum size required to lay out the container, taking into account the components in the specified container.

**Syntax**

public Dimension minimumLayoutSize(Container parent)

**Parameters**

***parent***

Container that needs to be laid out;

**Description**

This method calculates and returns the minimum dimensions required by the CardLayout manager to lay out the components contained within the specified container.

**Imports**

*import java.awt.CardLayout;*

**Returns**

The return type of this method is Dimension. This return value contains the minimum height and width required to lay out the container in the specified panel.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class and to the chapter project.

## next(Container)

**ClassName**

CardLayout

**Purpose**

To make the next card in the card stack visible. If the current card is the bottom-most card in the stack then the first card is made visible.

**Syntax**

public void next(Container parent)

**Parameters**

***parent***

The parent Container object that this CardLayout object is the lay out manager for.

**Description**

The card currently in view is hidden and the card just below it is displayed.

**Imports**

*import java.awt.CardLayout;*

**Returns**

None.

**Example**

The project at the end of this chapter uses this method.

## preferredLayoutSize(Container)

**ClassName**

CardLayout

**Purpose**

To calculate the preferred dimensions for this layout, taking into account the components in the specified container.

**Syntax**

public Dimension preferredLayoutSize(Container target)

**Parameters**

***target***

Container that needs to be laid out.

**Description**

This method computes the ideal width and height required by this layout. The values returned have no effect unless the program specifically enforces these dimensions.

**Imports**

*import java.awt.CardLayout;*

**Returns**

The return type of this method is Dimension. This return value contains the ideal height and width required to lay out the container in the specified panel.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## previous(Container)

**ClassName**

CardLayout

**Purpose**

To make the previous card in the card stack visible. The previous card to the top-most card in the stack is deemed to be the card at the bottom of the stack.

**Syntax**

public void previous(Container parent)

**Parameters**

***parent***

The parent Container object that this CardLayout object is the layout manager for

**Description**

The card currently in view is hidden and the card just above it is displayed.

**Imports**

*import java.awt.CardLayout;*

**Returns**

None.

**Example**

The project at the end of this chapter uses this method.

## removeLayoutComponent(Component)

**ClassName**

CardLayout

**Purpose**

To remove the specified component from the layout.

**Syntax**

public void removeLayoutComponent(Component comp)

**Parameters**

***comp***

Component to be removed from the layout.

**Description**

This method disassociates the component being removed from the named area to which it was added.

**Imports**

*import java.awt.CardLayout;*

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the example given under the corresponding function in the FlowLayout class.

## show(Container, String)

**ClassName**

CardLayout

**Purpose**

To make the specified card visible.

**Syntax**

public void show(Container parent, String name)

**Parameters**

***parent***

The parent Container object that this CardLayout object is the layout manager for.

***name***

Name of the card in the stack.

**Description**

The *name* parameter is specified when adding cards to the stack. This name is used to bring the specified card into view.

**Imports**

*import java.awt.CardLayout;*

**Returns**

None.

**Example**

The project at the end of this chapter uses this method.

## toString()

**ClassName**

CardLayout

**Purpose**

To represent the parameter values of the CardLayout object as a String.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method returns a String representation of this CardLayout object's values, namely, the horizontal and vertical gap to leave between components. Each value is prefixed with a short descriptive tag.

**Imports**

*import java.awt.CardLayout;*

**Returns**

This method returns a String containing the values of the CardLayout object, each value prefixed by a descriptive tag.

**See Also**

The toString method of the Object class

**Example**

Refer to the example given under the corresponding function in the FlowLayout class and to the chapter project.

## GridBagLayout

**Purpose**

A very sophisticated layout manager that can lay out individual components using different constraints, in a manner such that components can be of different sizes and can be aligned vertically and horizontally.

**Syntax**

public class GridBagLayout extends Object implements LayoutManager

**Description**

This is the most powerful of all the layout managers that the Abstract Windowing Toolkit provides. Consequently, it is also the most complex layout manager. This layout manager can lay out components on a grid of cells, with each component occupying one or more cells in the grid. The group of cells occupied by a component is known as its display area. A GridBagConstraints object is associated with each component and these constraints instruct the layout manager to lay out the component in a particular manner. Refer to the GridBagConstraints class in this chapter for more information on customizing the appearance of a Container using a GridBagLayout layout manager. Figure 4-15 shows the inheritance hierarchy for the GridBagLayout class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.GridBagLayout;*

**Constructors**

public GridBagLayout()

**Parameters**

None.

**Example**

Refer to the chapter project for a detailed example of a GridBagLayout layout manager

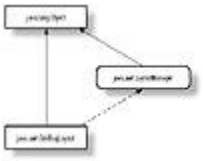

**Figure 4-15** Inheritance hierarchy for the GridBagLayout class

## addLayoutComponent(String, Component)

**ClassName**

GridBagLayout

**Purpose**

To add the specified component to the layout.

**Syntax**

public void addLayoutComponent(String name, Component comp)

**Parameters**

***name***

Name of the component to be added.

***comp***

Component object to be added.

**Description**

The GridBagLayout uses a GridBagConstraints object to lay out a component and does not associate a name with a component. In order to conform to the LayoutManager interface, this method exists but is an empty stub.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

Refer to the chapter project to see how components are added to a GridBagLayout and how a GridBagConstraints object instructs this layout manager how to lay out the particular component.

## AdjustForGravity(GridBagConstraints, Rectangle)

**ClassName**

GridBagLayout

**Purpose**

To modify the position and dimensions of the component depending on the specified constraints.

**Syntax**

protected void AdjustForGravity(GridBagConstraints c, Rectangle r)

**Parameters**

***c***

GridBagConstraints object specifying the constraints.

***r***

Rectangle specifying the coordinates and dimensions of a component.

**Description**

The coordinates of the rectangle *r* and the height and width dimensions are set according to the constraints, specified in the GridBagConstraints object *c*. The GridBagConstraints object specifies the geometry constraint, as well as any padding that is to be added to the size of the component.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

None.

**Example**

This method is a protected method in the GridBagLayout class and is used internally by the GridBagLayout implementation.

## ArrangeGrid(Container)

**ClassName**

GridBagLayout

**Purpose**

To lay out the components in the container.

**Syntax**

protected void ArrangeGrid(Container parent)

**Parameters**

***parent***

Container to be laid out.

**Description**

This protected method implements the layout policy of this layout manager and causes the components in the specified container to be laid out according to the constraints associated with each component.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

None.

**Example**

This method is a protected method in the GridBagLayout class and is used internally by the GridBagLayout implementation.

## DumpConstraints(GridBagConstraints)

**ClassName**

GridBagLayout

**Purpose**

To print the values of the specified GridBagConstraints object.

**Syntax**

protected void DumpConstraints(GridBagConstraints c)

**Parameters**

***c***

GridBagConstraints object whose values are to be printed on the screen.

**Description**

This method is useful for debugging the implementation of the GridBagLayout layout manager. It simply prints the values of the variables of the specified GridBagConstraints object.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

None.

**Example**

This method is a protected method in the GridBagLayout class and is only used for debugging purposes by the GridBagLayout implementation.

## DumpLayoutInfo(GridBagLayoutInfo)

**ClassName**

GridBagLayout

**Purpose**

To print debugging information contained in the layout parameters of the specified GridBagLayoutInfo object.

**Syntax**

protected void DumpLayoutInfo(GridBagLayoutInfo i)

**Parameters**

***i***

GridBagLayoutInfo object whose values are to be printed on the screen.

**Description**

The GridBagLayoutInfo class is used internally by the GridBagLayout class and it contains information about the layout (such as the number of cells horizontally and vertically in the layout, the largest minimum width in each row, and other parameters). This method simply prints these parameter values.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

None.

**Example**

This method is a protected method in the GridBagLayout class and is only used for debugging purposes by the GridBagLayout implementation.

## getConstraints(Component)

**ClassName**

GridBagLayout

**Purpose**

To get the GridBagConstraints object associated with the specified component.

**Syntax**

public GridBagConstraints getConstraints(Component comp)

**Parameters**

***comp***

Component object whose GridBagConstraints should be retrieved.

**Description**

This method returns a copy of the GridBagConstraints object associated with the specified component.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

The return value of this method is a GridBagConstraints object.

**See Also**

The GridBagConstraints class

**Example**

Please refer to the documentation on the GridBagConstraints class in this chapter.

## getLayoutDimensions()

**ClassName**

GridBagLayout

**Purpose**

To get the largest minimum width measurements of the components in each row and the largest minimum height measurements of the components in each column .

**Syntax**

public int [] [] getLayoutDimensions()

**Parameters**

None.

**Description**

This method returns the largest minimum width and height measurements of components in each row and column of the layout.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

This method returns a two-dimensional array of integers, where the widths are specified in the first row of integers and the heights are specified in the second row.

**Example**

Please refer to chapter project for an example.

## GetLayoutInfo(Container, int)

**ClassName**

GridBagLayout

**Purpose**

To determine the GridBagLayoutInfo parameters for the components in the specified container.

**Syntax**

protected GridBagLayoutInfo GetLayoutInfo(Container parent, int sizeflag)

**Parameters**

***parent***

Container object to be laid out.

**sizeflag**

Specifies whether to use each components' preferredSize method or minimumSize method, to determine the size of each component, while laying out the container.

**Description**

The GridBagLayoutInfo class is used internally by the GridBagLayout class and it contains information about the layout (such as the number of cells horizontally and vertically in the layout, the largest minimum width in each row, and other parameters). This method determines these parameter values.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

This method returns a GridBagLayoutInfo object containing the parameters for the current layout configuration.

**Example**

This method is a protected method in the GridBagLayout class and is only used internally by the GridBagLayout implementation.

## getLayoutOrigin()

**ClassName**

GridBagLayout

**Purpose**

To get the coordinates of the starting point of this layout.

**Syntax**

public Point getLayoutOrigin()

**Parameters**

None.

**Description**

This method returns the x and y coordinates of the origin of the layout.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

This method returns a Point object that specifies the origin of the layout.

**Example**

Please refer to chapter project for an example.

## getLayoutWeights()

**ClassName**

GridBagLayout

**Purpose**

To get the weights along each row and column.

**Syntax**

public double [] [] getLayoutWeights()

**Parameters**

None.

**Description**

This method returns the weights along each row and column of the layout. Weights are used to specify how to distribute space among the components of a row or column. Weights play an important role in the resizing behavior of a component. If the component is weighted in the direction of the *x* axis (GridBagConstraints.weightx), then the component will expand horizontally, and the component will expand vertically if it is weighted in the direction of the *y* axis (GridBagConstraints.weighty).

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

This method returns a two-dimensional array of double precision floating point numbers that specifies the weights of the components in each row and column.

**See Also**

The weight and weighty variables of the GridBagConstraints class

**Example**

Please refer to chapter project for an example.

## GetMinSize(Container, GridBagLayoutInfo)

**ClassName**

GridBagLayout

**Purpose**

To determine the minimum dimensions for the layout.

**Syntax**

protected Dimension GetMinSize(Container parent, GridBagLayoutInfo i)

**Parameters**

***parent***

Container object to be laid out.

***i***

GridBagLayoutInfo object containing the parameters of the overall layout configuration.

**Description**

This method is used to calculate the minimum width and height measurements required of the layout. The minimum dimensions are calculated according to the following formulae:
Minimum width = (Left + Right insets of *parent*) + (sum of largest minimum widths in each column)
Minimum height = (Top + Bottom insets of *parent*) + (sum of largest minimum heights in each row)

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

This method returns a Dimension object containing the minimum dimensions of the layout.

**See Also**

The minimumLayoutSize method of this class

**Example**

This method is a protected method in the GridBagLayout class and is only used internally by the GridBagLayout implementation.

## layoutContainer(Container)

**ClassName**

GridBagLayout

**Purpose**

To lay out the specified container according to the constraints for each component.

**Syntax**

public void layoutContainer(Container parent)

**Parameters**

***parent***

Container object to be laid out.

**Description**

This method lays out the components in the container, using the GridBagConstraints object associated with each component as a guideline for laying out the components. Applications do not directly invoke this method. The layout method of the Container class results in a call to this method.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

None.

**See Also**

The LayoutManager interface; the Container class

**Example**

Please refer to chapter project for an example.

## location(int, int)

**ClassName**

GridBagLayout

**Purpose**

To get the coordinates of the top left corner of the component where the specified point *x*, *y* lies.

**Syntax**

public Point location(int x, int y)

**Parameters**

***x***

The x coordinate of the point.

***y***

The y coordinate of the point.

**Description**

This method returns the location of the component containing the specified point.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

This method returns the coordinates of the component that contains the point specified by the given *x* and *y* values.

**Example**

Please refer to chapter project for an example

## lookupConstraints(Component)

**ClassName**

GridBagLayout

**Purpose**

To get the GridBagConstraints object associated with the specified component.

**Syntax**

protected GridBagConstraints lookupConstraints(Component comp)

**Parameters**

***comp***

Component object whose GridBagConstraints should be retrieved.

**Description**

This method retrieves the GridBagConstraints object associated with the specified component. The object returned by this method is the actual constraints object used by the GridBagLayout layout manager, and hence, care should be taken if one is modifying the parameters of this GridBagConstraints object.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

This method returns a GridBagConstraints object containing the constraints used for laying out the specified component.

**See Also**

The GridBagConstraints class

**Example**

This method is a protected method in the GridBagLayout class and is only used internally by the GridBagLayout implementation.

## minimumLayoutSize(Container)

**ClassName**

GridBagLayout

**Purpose**

To calculate the minimum size required to lay out the container, taking into account the components in the specified container.

**Syntax**

public Dimension minimumLayoutSize(Container parent)

**Parameters**

***parent***

Container that needs to be laid out.

**Description**

This method calculates and returns the minimum dimensions required by the GridBagLayout manager to lay out the components contained within the specified container.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

The return type of this method is Dimension. This return value contains the minimum height and width required to lay out the container in the specified panel.

**See Also**

The Container class; the LayoutManager interface

**Example**

This method can be invoked on a GridBagLayout object, similar to the manner in which it is invoked on the corresponding function in the FlowLayout class.

## preferredLayoutSize(Container)

**ClassName**

GridBagLayout

**Purpose**

To calculate the preferred dimensions for this layout, taking into account the components in the specified container.

**Syntax**

public Dimension preferredLayoutSize(Container target)

**Parameters**

***target***

Container that needs to be laid out.

**Description**

This method computes the ideal width and height required by this layout. The values returned have no effect unless the program specifically enforces these dimensions.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

The return type of this method is Dimension. This return value contains the ideal height and width required to lay out the container in the specified panel.

**See Also**

The Container class, the LayoutManager interface

**Example**

This method can be invoked on a GridBagLayout object, similar to the manner in which it is invoked on the corresponding function in the FlowLayout class.

## removeLayoutComponent(Component)

**ClassName**

GridBagLayout

**Purpose**

To remove the specified component from the layout.

**Syntax**

public void removeLayoutComponent(Component comp)

**Parameters**

***comp***

Component to be removed from the layout.

**Description**

This method is a dummy stub in the GridBagLayout class, as this layout manager doesn't need to maintain associations between components and areas on the display.

**Imports**

*import java.awt.GridBagLayout;*

**Returns**

None.

**See Also**

The Container class; the LayoutManager interface

**Example**

This method can be invoked on a GridBagLayout object, similar to the manner in which it is invoked on the corresponding function in the FlowLayout class.

## setConstraints(Component, GridBagConstraints)

**ClassName**
GridBagLayout
**Purpose**
To apply the GridBagConstraints to the specified component.
**Syntax**
public void setConstraints(Component comp, GridBagConstraints constraints)
**Parameters**
***comp***
Component object that the constraints are to be applied to.
**constraints**
Constraints for the component.
**Description**
This method applies the constraints specified in the *constraints* parameter to the specified component. These constraints are used to determine the position and dimensions of the component.
**Imports**
*import java.awt.GridBagLayout;*
**Returns**
None.
**See Also**
The GridBagConstraints class
**Example**
Please refer to the chapter project.

## toString()

**ClassName**
GridBagLayout
**Purpose**
To represent the values of the GridBagLayout object as a String.
**Syntax**
public String toString()
**Parameters**
None.
**Description**
As the parameters associated with each component may be many in number, this method just prints the classname of this object (java.awt.GridBagLayout)
**Imports**
*import java.awt.GridBagLayout;*
**Returns**
This method returns a String containing the name of the GridBagLayout object.
**See Also**
The toString method of the Object class
**Example**

This method can be invoked on a GridBagLayout object similar to the manner in which it is invoked on the corresponding function in the FlowLayout class.

## GridBagConstraints

**Purpose**

To specify the constraints for laying out a component using the GridBagLayout class.

**Syntax**

public class GridBagConstraints extends Object implements Cloneable

**Description**

The public variables of this class are used to specify the constraints for laying out a component within a container that uses a GridBagLayout object as its layout manager. Every component within the container is associated with an instance of this class. The GridBagConstraints values specify how the component is laid out within the container. Figure 4-16 shows the inheritance hierarchy for the GridBagConstraints class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.GridBagConstraints;*

**Constructors**

public GridBagConstraints()

**Parameters**

None.

**Variables**

The following are the public variables that can be accessed and modified directly from within an application.

**public int *anchor***

The value of this variable specifies where in the display area the GridBagLayout class will anchor this component if its display area is larger than the component. The point in the display area where the component can be anchored can be specified using one of the following values:
GridBagConstraints.NORTH
GridBagConstraints.SOUTH
GridBagConstraints.EAST
GridBagConstraints.WEST
GridBagConstraints.NORTHEAST
GridBagConstraints.NORTHWEST
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTHWEST
GridBagConstraints.CENTER
The default value is GridBagConstraints.CENTER.

**public Insets *insets***

The top, bottom, left, and right padding to leave between the component and the edge of its display area.

**public int *ipadx***

The number of pixels to pad on the left and right sides of the component. Twice this value is added when calculating the minimum size for this component.

**public int *ipady***

The number of pixels to pad on the top and bottom sides of the component. Twice this value is added when calculating the minimum size for this component.

**public int *gridx***

Row number of the cell that occupies the upper-left corner of the component's display area. Setting this value to GridBagConstraints.RELATIVE instructs the GridBagLayout class to lay out this component to the right of the previously added component.

**public int *gridy***

Column number of the cell at the upper-left corner of the display area. Setting this value to GridBagConstraints.RELATIVE instructs the GridBagLayout class to lay out this component just below the previously added component.

**public int *gridwidth***

Width of the component's display area expressed as a number of cells in a row. Setting this value to GridBagConstraints.REMAINDER instructs the GridBagLayout class that this component is the last in its row. Setting this value to GridBagConstraints.RELATIVE instructs the GridBagLayout class that this component is next to the last in its row. The default value for this variable is 1.

**public int *gridheight***

Height of the component's display area expressed as a number of cells in a column. Setting this value to GridBagConstraints.REMAINDER instructs the GridBagLayout class that this component is the last in its column. Setting this value to GridBagConstraints.RELATIVE instructs the GridBagLayout class that this component is next to the last in its column. The default value for this variable is 1.

**public double *weightx***

Specifies whether or not the component's width should increase if it needs to be resized. A value must be specified for at least one component in a row. The default value for this variable is 0.

**public double *weighty***

Specifies whether or not the component's height should increase if it needs to be resized. A value must be specified for at least one component in a column. The default value for this variable is 0.

**Example**

Refer to the section project for a detailed example that uses this class.

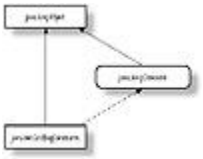

**Figure 4-16** Inheritance hierarchy for the GridBagConstraints class

## clone()

**ClassName**

GridBagConstraints

**Purpose**

To create a duplicate of this GridBagConstraints object

**Syntax**

public Object clone()

**Parameters**

None.

**Description**

A new instance of a GridBagConstraints object is created and an exact duplicate of this GridBagConstraints is made.

**Imports**

*import java.awt.GridBagConstraints;*

**Returns**

The return value is an Object that is a clone of this GridBagConstraints object. This return value must be cast as a GridBagConstraints object in order to use it as one.

**See Also**

The clone method of the Object class; the Cloneable interface

**Example**

The following sample code demonstrates the implementation of this method.

```
import java.awt.GridBagConstraints;

public class GBCTest {
    public static void main(String args[]) {
        // GridBagConstraints constructor
        GridBagConstraints g1 = new GridBagConstraints();
        g1.gridx = 5;
        g1.gridy = 4;
        GridBagConstraints g2 = (GridBagConstraints)g1.clone();
        System.out.println("g1.gridx = " + g1.gridx);
        System.out.println("g2.gridx = " + g2.gridx);
    }
}
```

## The Layout Demonstration

By now you are familiar with the layout manager interface and the API of Java's ready-made layout managers. In the following project we will build an application that demonstrates the functionality of each of the layout managers discussed in this chapter. The project will help you visualize the effect that each of the layout managers: FlowLayout, GridLayout, BorderLayout, CardLayout, and GridBagLayout have on the appearance of the Container. Figure 4-17 shows a screenshot of this project.

**Figure 4-17** The Layout Demonstration project

In this project, we present a simple user interface that allows people to see the impact of each particular layout manager by selecting from a list of choices. The appearance and position of the components on the screen change with the selection of a new layout. In this project, a set of buttons is created and these buttons are placed within a Panel using different layout techniques. The simple user interface lets you flip through different layout views. This project also demonstrates customizing a layout by specifying parameters (such as the alignment mode, the horizontal gap between components, and so on).

## Assembling the Project

**1.** Create and edit a file named LayoutDemo.java and use this file to enter the code for this project. First, ensure that the necessary Java modules are imported.

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

**2.** Now create a panel containing buttons. The SlidePanel class does this. The buttons on this panel will be laid out using different layout managers. By passing a different layout manager as the argument to the methods of the SlidePanel class, we can create different layouts of the buttons created by the SlidePanel class. This is accomplished by associating a layout manager with this Panel. The buttons are added to different areas of the panel. The significance of these areas depends on the layout manager being used to lay out the buttons on the panel. A Label component at the bottom of the panel displays the parameters of the layout manager used to lay out the components on the panel.

```
class SlidePanel extends Panel {
    Panel newSlide(LayoutManager layout) {
        Panel parent = new Panel();
        parent.setLayout(new BorderLayout());
        // print the layout manager parameters in a Label
component
        String s = new String("Layout parameters: ");
        s += layout.toString();
        parent.add("South", new Label(s)); // attach the label at
          the bottom
        Panel p = new Panel();
        p.setLayout(layout);
        p.add("North", new Button("This"));
        p.add("West", new Button("demonstration"));
        p.add("Center", new Button("is"));
        p.add("East", new Button("really"));
        p.add("South", new Button("cool !"));
        parent.add("Center", p);
        return parent;
}
```

**3.** The GridBagLayout is a complex layout manager that requires constraints to be associated with each component. Add this method to the SlidePanel class. It arranges the buttons on the SlidePanel using the GridBagLayout class. The actual assignment of constraints to buttons and adding them to the layout is implemented as three protected methods: makeFirstRow, makeSecondRow, and

makeThirdRow. A Label component at the bottom of the panel displays the parameters of the layout manager used to lay out the components on the panel.

```
// use the GridBagLayout class to layout buttons
Panel gridBagSlide() {
        Panel parent = new Panel();
        parent.setLayout(new BorderLayout());
        Panel p = new Panel();
        GridBagLayout gbl = new GridBagLayout();
        String s = new String("Layout parameters: ");
        s += gbl.toString();
        parent.add("South", new Label(s)); // attach the label at
          the bottom
        p.setLayout(gbl);
        GridBagConstraints gbc = new GridBagConstraints();

        makeFirstRow(gbc, gbl, p);         // assign constraints
                                              to buttons
        makeSecondRow(gbc, gbl, p);        // row by row
        makeThirdRow(gbc, gbl, p);         // add the entire
panel
                                              to the parent
        parent.add("Center", p);
        return parent;
}
```

**4.** Assign constraints such that three buttons are laid out on the first row. Assigning a value to weightx of the GridBagConstraints object ensures that the buttons will expand horizontally and occupy space along the row, if the window is resized.

```
    protected void makeFirstRow(GridBagConstraints gc,
      GridBagLayout gl, Panel pan) {
        // create 2 buttons of width 1 cell each on the same row
        gc.gridx = GridBagConstraints.RELATIVE;
        gc.fill = GridBagConstraints.BOTH;
        gc.weightx = 1.0;
        Button b1 = new Button("Buttons");
        gl.setConstraints(b1, gc);
        pan.add(b1);
        Button b2 = new Button("in a");
        gl.setConstraints(b2, gc);
        pan.add(b2);
        // let this button be the last in this row
        gc.gridwidth = GridBagConstraints.REMAINDER;
        Button b3 = new Button("row");
        gl.setConstraints(b3, gc);
        pan.add(b3);
    }
```

**5.** Create and lay out a button such that it occupies all the remaining space in a row. In addition, add a second button that occupies two rows.

```
    protected void makeSecondRow(GridBagConstraints gc,
      GridBagLayout gl, Panel pan) {
        // create a button on the next row
        gc.gridwidth = GridBagConstraints.RELATIVE;
        gc.gridheight = 1;
        gc.weightx = 0.0;
        gc.weighty = 0.0;
```

```
        Button b4 = new Button("PressMe");
        gl.setConstraints(b4, gc);
        pan.add(b4);

        // create a large button that will expand heightwise if
           resized
        gc.gridwidth = GridBagConstraints.REMAINDER;
        gc.gridheight = 2;
        gc.weightx = 0.0;
        gc.weighty = 1.0;
        Button b5 = new Button("Large Button");
        gl.setConstraints(b5, gc);
        pan.add(b5);
    }
```

**6.** On the third row, set the constraints so that a button occupies the entire row.

```
    protected void makeThirdRow(GridBagConstraints gc,
      GridBagLayout gl, Panel pan) {
        // let this button take up the entire row
        gc.gridwidth = GridBagConstraints.REMAINDER;
        gc.gridheight = 1;
        gc.weightx = 0.0;
        gc.weighty = 0.0;
        Button b6 = new Button("Long Button");
        gl.setConstraints(b6, gc);
        pan.add(b6);
    }
```

**7.** The main display needs to be able to display multiple views of the Panel of buttons, each view implementing a particular layout. The CardLayout layout manager is ideal for this task. Each view can be a card in the card layout. To display a layout, the card containing the view has to be made visible. In the constructor for the SlidePanel class, create new Panels of buttons and associate each with a different layout manager. The SlidePanel class is now complete.

```
    SlidePanel() {
        setLayout(new CardLayout());
        // add each slide as a new card and give each slide a
name
        add("FlowLayout1", newSlide(new FlowLayout()));
        add("FlowLayout2", newSlide(new FlowLayout
          (FlowLayout.LEFT)));
        add("FlowLayout3",
        newSlide(new FlowLayout(FlowLayout.LEFT, 30, 0)));
        add("GridLayout1", newSlide(new GridLayout(1, 0)));
        add("GridLayout2", newSlide(new GridLayout(3, 2)));
        add("GridLayout3", newSlide(new GridLayout(3, 2, 20,
10)));
        add("BorderLayout1", newSlide(new BorderLayout()));
        add("BorderLayout2", newSlide(new BorderLayout(10, 20)));
        add("GridBagLayout", gridBagSlide());
    }
}
```

**8.** Now create the LayoutDemo class. For this class to run as an applet, it will have to extend the java.applet.Applet class. It maintains a reference to the card stack of panels and shuffles this card stack to display the various layout views. The display area that will contain the card stack of panels is positioned above the

choice control with which you can change the layout being used to place the buttons on the screen. Using the Center and South areas of a BorderLayout accomplishes this in a snap. A Panel is used to neatly arrange the choice component and the layout description label. A Choice component, with the names of all the different layouts that can be viewed, is added to the Panel. This code is implemented in the init method of the LayoutDemo class. The following sample code is an example of what it takes to implement this portion of the project.

```
public class LayoutDemo extends Applet {
    SlidePanel viewStack;          // stack of panels
    public void init() {
        setLayout(new BorderLayout());
        viewStack = new SlidePanel();  // stack of layout views
        Label l = new Label("Using CardLayout to view other
          layouts...");
        add("North", l);
        add("Center", viewStack);
        Panel p = new Panel();
        p.setLayout(new FlowLayout(FlowLayout.LEFT));
        add("South", p);
        Choice c = new Choice();       // add the various layout
                                          options
        c.additem("FlowLayout1");        // that can be viewed
        c.addItem("FlowLayout2");
        c.addItem("FlowLayout3");
        c.addItem("GridLayout1");
        c.addItem("GridLayout2");
        c.addItem("GridLayout3");
        c.addItem("BorderLayout1");
        c.addItem("BorderLayout2");
        c.addItem("GridBagLayout");
        p.add(c);                     // add the Choice component
                                          to the Panel

        // create and add buttons for the user to flip through
the
           cards
        p.add(new Button("First card"));
        p.add(new Button("Last card"));
        p.add(new Button("Next card"));
        p.add(new Button("Previous card"));
    }
```

All that remains to be implemented is a simple interface for the applet. Using this interface you can view the different layouts.

**9.** The following event handler determines which of the controls was activated by the user and displays the corresponding card. The event handler brings the card, corresponding to the selected choice, to the top of the view stack.

```
public boolean action(Event evt, Object arg) {
    if (evt.target instanceof Choice) { // display the choice
     selected
                    ((CardLayout)viewStack.getLayout()).
                      show(viewStack,(String)arg);
    } else if ("First card".equals(arg)) { // display the first
card
```

```
                    ((CardLayout)viewStack.getLayout()).
                      first(viewStack);
    } else if ("Last card".equals(arg)) {  // display the last
card
                    ((CardLayout)viewStack.getLayout()).
                      last(viewStack);
    } else if ("Next card".equals(arg)) {  // display the next
card
                    ((CardLayout)viewStack.getLayout()).
                      next(viewStack);
    } else if ("Previous card".equals(arg)) { // display the
      previous card
                    ((CardLayout)viewStack.getLayout()).
                      previous(viewStack);
    }
     return true;
}
```

**10.** And now the final step of creating a main function, required to launch the application if it were executed as a stand-alone Java application. It creates a top-level Frame window and emulates the behavior of an applet, by invoking the init methods of the LayoutDemo class.

```
    public static void main(String args[]) {
        Frame f = new Frame("Layout Demonstration");
        LayoutDemo ld = new LayoutDemo();
        ld.init();
        ld.start();

        f.add("Center", ld);
        f.resize(450, 300);
        f.show();
    }
}
```

And we are done!

**11.** Save the LayoutDemo.java file and compile the project by executing the following command:

```
javac LayoutDemo.java
```

**12.** Now run the program by executing the following command:

```
java LayoutDemo
```

## How It Works

The Layout Demonstration project used different layout managers to lay out the same set of buttons on the screen. The CardLayout layout manager was used to present the different layout views, one at a time, to the user. Panels and Frames were used as containers for components (such as Buttons and Labels). Choice components and buttons that initiated actions enabled the user to flip through the various views in the CardLayout. The ready-made layout managers that are in the Java AWT are sufficient for most applications. Choose the layout manager or windowing component most suited to your requirements. By nesting panels within panels, you can use different layout managers for different parts of your user interface.

Have fun in creating user interfaces for Java applications!

### setMenuBar(MenuBar)

**ClassName**
Frame
**Purpose**
Sets the menu bar for this Frame to the specified MenuBar object
**Syntax**
public synchronized void setMenuBar(MenuBar mb)
**Parameters**
**mb**
The MenuBar object that represents the menu bar for this Frame
**Description**
This method specifies the menu bar to use on this Frame window.
**Imports**
*import java.awt.Frame;*
**Returns**
None.
**See Also**
The Image Class
**Example**
Refer to the examples in the MenuBar section of Chapter 6.

# Chapter 5
# Handling Text, Dialogs, And Lists

An application might need input not only at the beginning of execution but at various stages of a run. The input may not be from a known list of inputs or even if it is, the number of items might be large. It may be necessary to update the user about the state of the application at various stages and to obtain his approval before performing the successive stages. This chapter introduces the properties and use of text handling components, dialog boxes that "converse" with users, and scrolling lists of items that offer a large number of choices. We will cover the classes TextComponent, TextArea, TextField, Dialog, FileDialog and List and describe their methods in detail. The application developed at the end of this chapter is a basic framework for an API Reference Interface Application. Using this interface, users can specify a class name and obtain information about any number of methods in the specified class. They can view the details of any method and optionally can save them in a file.

## Handling Text

Depending on an application's charateristics, text input from the user can be a single character, single word, single line, or multiple lines of text. Handling such varying input types is important for a smooth-running application. In Java, the TextComponent, TextArea, and TextField classes in the AWT package provide the necessary interface components for text handling. TextArea and TextFields are subclasses of the TextComponent class. A TextField accepts a single line of text. Consider an application that handles e-mail. You know that e-mail addresses are not multiline text; you can use instances of TextField for the To, Cc, and Subject fields, but for the body of your e-mail message you need a multiline editor.

The editing capabilities of TextField and TextArea are the same. You can disallow editing in both. You can point to any location in the area and enter the input. In a TextField, you have a single line of boxed text visible to you. The number of columns in the text field is application specific, but you can type as many charaters as you want and the text will move to the left accommodating more input. To view the text, you have to move the cursor to the desired location. A TextArea provides two scrollbars (vertical and horizontal) for viewing different parts of the TextArea and editing with ease. Figure 5-1 shows an example To TextField. The second text field, Cc, contains the sender's name as a default string which saves the time required to type in an e-mail id.

**Figure 5-1** A single-line TextField component

The text area, allowing multiple lines to be edited, is created using the TextArea class in the AWT package. Figure 5-2 illustrates the text area provided to edit the body of an e-mail message. It includes scrollbars that allow you to go back and forth in editing. If a certain text field or text area should be protected from editing, you can disable it. Also, in situations where you don't want the characters you type to appear on the screen, such as entering a password, TextField can set echo characters that appear on the screen for each character you enter. Figure 5-3 shows an example in which the user entry is masked by echo characters.

**Figure 5-2** A multiline TextArea component

**Figure 5-3** Echo characters in a TextField

## Dialogs in Java

A primary window is the root window from which all the other windows used by an application are generated. In the case of Java GUIs, it is a Frame object for an application

and a Window for an applet. Applications use dialog windows to conduct context-specific dialog with the user. When a dialog window is closed, its parent is not affected. But the input given through the dialog window is available to the parent even after the dialog window is closed. In Java, two classes implement dialog windows: Dialog and FileDialog. The class Dialog implements a pop-up window to interact with the user. You can design it as a simple prompt window, a message window, or an input window and so on. Depending on the input characteristic, you can specify the Dialog window to be modal or non-modal. A modal Dialog box prevents any action on other windows of the application until the user responds to the Dialog box with some input. If you specify a Dialog box to be non-modal, the user can work on other windows without entering any input for the dialog window. Figure 5-4 shows a Dialog window instance created using the JDK. The Dialog box provides two buttons: OK and Cancel. Selecting either of them decides the next step of the application.


**Figure 5-4** A sample dialog box in Java

FileDialog is a Java class that implements a pop-up window offering a selection of files to the user. This type of window can be created in either LOAD or SAVE mode. If in LOAD mode, the FileDialog window is created and an Open button is provided. In SAVE mode, a Save button appears in the file selection window. But you (as a programmer) are responsible for handling the loading and saving of files. You can get the name of the selected file from the FileDialog window. For example, someone might want to include a particular file into an e-mail. Figure 5-5 shows the use of a FileDialog component where directory "chp7" is opened and it contains three subdirectories. By successively selecting the folders you end up with the directory in which the desired file resides. On selecting the file and by clicking the Open button, the file is selected using the given FileDialog component.

**Figure 5-5** Using a FileDialog component for loading a file

## Lists

Neither menus, checkboxes, nor pull-down menus are adequate when you need an interface to handle a large number of available options. A scrolling list of selectable items makes an efficient way to save window space and present a lot of items for selection. In Java, the class List encapsulates the required behavior of scrolling lists. You can set a List object to allow only one selection or multiple selections. In the case of a single-selection List, selecting an item automatically deselects any other item already selected in the list. In a multiple-selection List, the user can select any number of items from the

List. Figure 5-6 shows a list that allows multiple selections. The List class provides methods that support selecting items and manipulating the items selected.


**Figure 5-6** A List component with multiple selections enabled

## Text, Dialog, and List Class Summaries

Table 5-1 summarizes the classes necessary for developing user interfaces in Java using text, dialogs, and lists.

**Table 5-1** Class description for text, dialog, and list components

| Class Name | Description |
|---|---|
| TextComponent | A component that allows the editing of text. Forms the super class for TextArea and TextField. |
| TextArea | Provides an area in which to display several lines of text. The text can be either read-only or edited. |
| TextField | A single-line editor and a subclass of TextComponent. |
| Dialog | A window that takes input from the user. |
| FileDialog | A modal Dialog window displaying a file selection dialog. |
| List | A component that provides a scrolling list of text items from which the user can select one or many items. |

### TextComponent

**Purpose**

A component that allows the editing of text. Forms the super class for TextArea and TextField.

**Syntax**

public class TextComponent extends Component

**Description**

TextComponent is used to implement the window components involved in text editing. It forms the super class of all text related components.Hence, TextArea

and TextField are subclasses of this class. Methods of the TextComponent class allow selection of text, manipulating the selected text, and specifying a text component as either editable or read-only. This class has no public constructors. Figure 5-7 illustrates the inheritance relationship of the TextComponent class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.TextComponent;*

**Constructors**

None.

**Parameters**

None.

**Example**

The textDemo class, implemented in the following example (Listing 5-1), uses the textPanel class to illustrate the usage of methods in the classes TextComponent, TextArea, and TextField. This application uses all the methods in these classes. The user can type his name, which is echoed as asterisk and then press the ChangeText button and see the effect. Pressing CloneTextArea will make a new image of TextArea. Figure 5-8 shows the resultant window.

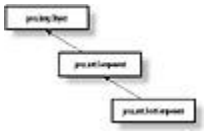

**Figure 5-7** Class diagram of TextComponent class

**Figure 5-8** The textDemo application in action

**Listing 5-1** textDemo.java: Program demonstrating the usage of methods in TextComponent, TextArea, and TextField

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

/**
        Filename: textDemo.java
        classes: textDemo
                 textPanel

        Purpose: demonstrating the usage of methods in the classes:
                        TextComponent, TextArea and TextField
*/

public class textDemo extends Frame {

        TextPanel txt_p;
                // to demonstrate textarea, textfield classes
```

```
        Menubar mbar;

        public textDemo() {

                // panel containing text components
                txt_p = new TextPanel();

                // menu to exit from the application
                mbar = new MenuBar();
                Menu quit = new Menu(“Quit”);
                quit.add(new MenuItem(“Stop”));
                mbar.add(quit);
                setMenuBar(mbar);
                        // add the panel to the North
                add(“North", txt_p);

                pack();
                show();

             }

        public boolean action(Event evt, Object arg) {

                // quit menu handler
                if (evt.target instanceof MenuItem) {
                        System.exit(0);
                        return true;
                }
                return false;

        }

              public static void main(String args[]) {

                textDemo txt_win = new textDemo();
                txt_win.setTitle(“Text Demo”);
                txt_win.pack();
                txt_win.show();
        }

}  // end of class textDemo

class TextPanel extends Panel {
      TextField name_f;        // text field to get "name"
 // panel that contains text field and area
        Panel txt_p;
               // text area to display manipulated string
      TextArea txt_edit;

      public TextPanel() {
        txt_p = new Panel();
        setLayout(new BorderLayout());

                name_f = new TextField(“Your Name", 15);
        if (!name_f.echoCharIsSet()) {
                name_f.setEchoCharacter('*');
```

```
                System.out.print(" The echo char is " );
System.out.println(name_f.getEchoChar());
        }
                Panel bot_p = new Panel();
                Button name_b = new Button("ChangeText");
                bot_p.add(name_f);
                bot_p.add(name_b);
                Button alt_txt = new Button("CloneTextArea");
                bot_p.add(alt_txt);

         txt_edit = new TextArea( 6,25);
               //6 rows, 25 columns
         txt_edit.setText("Enter Text Here");
         txt_p.add(txt_edit);

         add("North",txt_p);
         add("South", bot_p);

         show();

        }

        TextArea newArea;   // new text area for "cloning"

        public boolean action(Event evt, Object arg) {

            if (evt.target instanceof Button) {
                   // if the ChangeText button is pressed do:

            if ("ChangeText".equals(arg)){
                  String n_str = name_f.getText();
                  if (n_str.equals("Your Name"))
                            n_str = "";

                  txt_edit.selectAll();
                  int start =txt_edit.getSelectionStart();
                  int end   = txt_edit.getSelectionEnd();

                       String sel_t = txt_edit.getSelectedText();
            System.out.println(" selected text is " + sel_t);
                txt_edit.replaceText("Hello! ",0,sel_t.length());
                    txt_edit.appendText(" How are you?");
                    txt_edit.insertText(n_str, 7);

            // demonstrating the usage of following methods
                    System.out.println(" Name field has " +
                        name_f.getColumns() + "columns ");
                    System.out.println(" Min size of
                        name field is " +
                        name_f.minimumSize().width + " " +
                        name_f.minimumSize().height);
                    System.out.println(" Preferred size of
                        name field is " +
                       name_f.preferredSize().width + " " +
                       name_f.preferredSize().height);
                    System.out.println(" Min size of name
                         field with 10 rows is " +
```

```
                        name_f.minimumSize(10).width );
                    System.out.println(" Preferred size of
                        name field with 10 rows is " +
                        name_f.preferredSize(10).width + " "+
                        name_f.preferredSize(10).height);
    }

                   // if "CloneTextArea" button is pressed

    if ("CloneTextArea".equals(arg)) {
     System.out.println("CloneTextArea selected");
     int t_rows, t_cols;
     t_rows = txt_edit.getRows();
     t_cols = txt_edit.getColumns();
    String t_txt = txt_edit.getText();
    newArea = new
                TextArea(t_txt.getText(),t_rows,t_cols);

     newArea.setEditable(false);
    if (!newArea.isEditable())
     System.out.println(" the New are is NOT editable");
        Panel p = new Panel();
        p.add(newArea);

     txt_p.add("Center",p);
        System.out.println(" Min size of text area is " +
            txt_edit.minimumSize().width +
            " " + txt_edit.minimumSize().height);
        System.out.println(" Preferred size of text area
            is " + txt_edit.preferredSize().width + " " +
            txt_edit.preferredSize().height);
        System.out.println(" Min size of newtext area is "+
            newArea.minimumSize(10,25).width + " " +
            newArea.minimumSize(10,25).height);
        System.out.println(" Preferred size of newtext area
            is " + newArea.preferredSize(10,25).width +
            " " + newArea.preferredSize(10,25).height);

        Object fr = evt.target;
             //obtain the parent frame if you dont have handle
             // this illustrates a way to obtain frame handle

             while (fr!=null && !(fr instanceof Frame))
                fr = ((Component)fr).getParent();
             ((Frame)fr).pack();
             ((Frame)fr).show();
        }
      }
                return true;
    }

}  // end of textPanel class
```

## getSelectedText()

**ClassName**

TextComponent

**Purpose**

To get the text selected in the target TextComponent object.

**Syntax**

public String getSelectedText()

**Parameters**

None.

**Description**

Users can select text between desired locations or all of the text in a text component. This method obtains the text selected by the user for further manipulation according to the application's characteristics. It returns null if nothing is selected in the text component.

**Imports**

*import java.awt.TextComponent;*

**Returns**

The selected text contained in the TextComponent; the return type is String.

**See Also**

The TextArea class; the TextField class; the setText() method of the TextComponent class

**Example**

Refer to Listing 5-1. In textPanel class, this method is used to obtain the text in the text area as a value for the variable sel_t in the action method.

## getSelectionEnd()

**ClassName**

TextComponent

**Purpose**

To obtain the end index of the selected text.

**Syntax**

public int getSelectionEnd()

**Parameters**

None.

**Description**

Users can select text between desired locations or all of the text in a text component. This method obtains the end index of the selected text contained in the target TextComponent object.

**Imports**

*import java.awt.TextComponent;*

**Returns**

Returns the index position of the last character in the selected text; return type is int.

**See Also**

The TextArea and TextField classes; the setText() and getSelectedText() methods of the TextComponent class

**Example**

Refer to Listing 5-1. After selecting text in the text area using getSelectedText, this method is used in the textPanel class under the action() method to obtain the end index of selected text.

## getSelectionStart()

**ClassName**

TextComponent

**Purpose**

To obtain the start index of the selected text.

**Syntax**

public int getSelectionStart()

**Parameters**

None.

**Description**

User can select text between desired locations or all of the text in a text component. This method obtains the start index of the selected text contained in the target TextComponent object.

**Imports**

*import java.awt.TextComponent;*

**Returns**

The index position of the first character in the selected text. Return type is int.

**See Also**

The setText and getSelectedText methods of the TextComponent class; the TextArea and TextField classes

**Example**

Refer to Listing 5-1. After selecting the text in the text area using getSelectedText, this method is used to obtain the start index of the selected text in the action method of the textPanel class

## getText()

**ClassName**

TextComponent

**Purpose**

To obtain the text contained in the target TextComponent object.

**Syntax**

public String getText()

**Parameters**

None.

**Description**

TextComponent contains text and it can be edited if allowed. This method obtains the text contained in the text component. It is equivalent to selecting all the text and then getting that selected text.

**Imports**

*import java.awt.TextComponent;*

**Returns**

The text contained in the TextComponent is returned and the return type is String.

**See Also**

The setText method in theTextComponent class; the TextArea and TextField classes

**Example**

Refer to Listing 5-1. The name string entered in the text field is found by using this method to include the name (variable name_f) in the string to be written into the text area in the action() method in the textPanel class.

## isEditable()

**ClassName**

TextComponent

**Purpose**

To obtain the boolean value indicating whether the target TextComponent is editable or not.

**Syntax**

public boolean isEditable()

**Parameters**

None.

**Description**

Users can edit the text contained in a text component if the component is set to be editable using the setEditable method in the TextComponent class. This method finds out whether the text component is editable. It returns true if the text is editable; false if it is not editable.

**Imports**

*import java.awt.TextComponent;*

**Returns**

The boolean value indicating whether the text component is editable.

**See Also**

The setEditable method of theTextComponent class; theTextArea and TextField classes

**Example**

Refer to Listing 5-1. In the action method in class TextPanel, when CloneTextArea button is pressed, newArea is created and is set to disallow editing. This method is used to confirm its mode.

## paramString()

**ClassName**

TextComponent

**Purpose**

To obtain the parameter String of the target TextComponent object.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

Returns the String representation of the target TextComponent object, which contains the text. This method is protected and hence, can be used only by the classes within the *java.awt*package.

**Imports**

*import java.awt.TextComponent;*

**Returns**

The parameter string of type String.

**See Also**

The TextComponent and Component classes

**Example**

The following code uses paramString by subclassing the TextComponent.

```
package java.awt;
import java.awt.TextComponent;

class myText extends TextComponent {

        String myStringForm;

        public myText() {
         super("");
        }

         public String getmyStringForm() {
                         return super.paramString();
         }

         public static void main(String[] args) {
               myText txt = new myText();
               txt.getmyStringForm();
         }
}
```

## removeNotify()

**ClassName**

TextComponent

**Purpose**

To remove the peer of this text component.

**Syntax**

public void removeNotify()

**Parameters**

None.

**Description**

A text component peer is used to change the appearance of your text component, without changing its functionality. This method removes the peer of the target component.

**Imports**

*import java.awt.TextComponent;*

**Returns**

None.

**See Also**

The addNotify method of subclasses of TextComponent, namely TextArea and TextField; the TextComponentPeer class

**Example**

Refer to the details and information in the Chapter 9 describing Peers and manipulating peer interfaces.

## select(*int, int*)

**ClassName**

TextComponent

**Purpose**

Selects the text between the specified positions in the TextComponent.

**Syntax**

public void select(int start, int end)

**Parameters**

***start***

Index indicating the starting position of the selected text.

***end***

Index indicating the end position of the selected text.

**Description**

The textComponent contains a text. The user can select a part of the text or the entire text. This method selects the text contents between the two specified start and end positions. If the value of start is greater than end, then no text is selected.

**Imports**

*import java.awt.TextComponent;*

**Returns**

None.

**See Also**

The TextArea and TextField classes

**Example**

Refer to Listing 5-1. This method is used in the action method of class textPanel to obtain the check string after selecting the text or after getting the start and end position if the whole text is selected.

## selectAll()

**ClassName**

TextComponent

**Purpose**

Selects all of the text contained in the TextComponent.

**Syntax**

public void selectAll()

**Parameters**

None.

**Description**

TextComponent contains text. User can select any part of the text or the full text. This method selects all of the text contained in the text component.

**Imports**

*import java.awt.TextComponent;*

**Returns**

None.

**See Also**

The TextArea and TextField classes

**Example**

Refer to Listing 5-1. In the action method in the textPanel class; all the text in the text area is selected and the string is stored in sel_t. This string is used to change the contents of the text area.

## setEditable(*boolean*)

**ClassName**

TextComponent

**Purpose**

The boolean value, indicating whether the target TextComponent should or should not be editable, is set.

**Syntax**

public void setEditable(boolean okToEdit)

**Parameters**

***okToEdit***

The text component is editable if this is true; not editable if this is false.

**Description**

The user can edit the text contained in a text component if the component is set to be editable using this setEditable method in class TextComponent. To set the TextComponent to be editable, the boolean parameter should be true. To set it to be noneditable, the parameter should be false.

**Imports**

*import java.awt.TextComponent;*

**Returns**

None.

**See Also**

The isEditable method of the TextComponent class; the TextArea and TextField classes

**Example**

Refer to Listing 5-1, the newArea; the new text area formed is set to noneditable mode using this method. This occurs in the action method of the textPanel class.

## setText(*String*)

**ClassName**

TextComponent

**Purpose**

Sets the specified text to be the contents of the target TextComponent object.

**Syntax**

public void setText(String new_text)

**Parameters**

***new_text***

The text of type String which is to be the new text content of the TextComponent.

**Description**

TextComponent contains text which can be edited if permitted. This method sets the specified text to be the text content of the TextComponent. If the component did not contain any text prior to this method call, then the specified text is set to be the text. If it did contain text earlier, this method replaces the previous text with this new one.

**Imports**

*import java.awt.TextComponent;*

**Returns**

None.

**See Also**

The getText method of the TextComponent class; the TextArea and TextField classes

**Example**

Refer to Listing 5-1. In the constructor of textPanel class, this method is used to set the string in the text field to "Enter text here".

## TextArea

**Purpose**

Provides an area in which to display several lines of text. The text can either be read-only or read and edit.

**Syntax**

public class TextArea extends TextComponent

**Description**

The TextArea class provides an area in which to display several lines of text or allow editing of that text. So it can be considered to be a multiline editor, if editing is allowed. Wordwrap is set to true and both horizontal and vertical scrollbars are visible. If editing text is allowed, then any position in the text area can be reached using the mouse or the arrow keys. This allows users to move through the text as they would in a full-blown editor. TextArea is a subclass of the TextComponent class. Methods of TextComponent class that allow selection and manipulation of text can be performed in an editable text area. The setEditable method of TextComponent can be used to allow editing a TextComponent object. Figure 5-9 illustrates the inheritance relationship of the TextArea class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.TextArea;*

**Constructors**

public TextArea()
public TextArea(int t_rows, int t_cols)
public TextArea(String text)
public TextArea(String text, int t_rows, int t_cols)

**Parameters**

***t_rows***

The number of rows specified in the TextArea.

***t_cols***

The number of columns specified in the TextArea.

***text***

The text which forms the initial text contents of this TextArea.

**Example**

Refer to Listing 5-1. In the textPanel class, an instance of this class (TextArea) is a member of the class. It is constructed with 6 rows and 25 columns.

**Figure 5-9** Class diagram of the TextArea class

## addNotify()

**ClassName**

TextArea

**Purpose**

This method creates a peer of the target TextArea object.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

Creates an instance of the TextAreaPeer as a peer for the target TextArea object. Using the peer, you can change the appearance of the TextArea without modifying its functionality. This method is required if you are writing your own AWT.

**Imports**

*import java.awt.TextArea;*

**Returns**

None.

**See Also**

The TextAreaPeer class

**Example**

Refer to Chapter 9, which describes the peers and interface for details.

## appendText(*String*)

**ClassName**

TextArea

**Purpose**

Append the specified text to the text content of the target TextArea object.

**Syntax**

public void appendText(String add_text)

**Parameters**

***add_text***

The specified text of type String to be appended to the TextArea.

**Description**

This method appends the specified string to the text contained in the target TextArea object.

**Imports**

*import java.awt.TextArea;*

**Returns**

None.

**See Also**

The insertText method of the TextArea class; the TextArea and TextField classes

**Example**

Refer to Listing 5-1. The text "How are you?" is appended to the text area in the action method of the textPanel class.

## getColumns()

**ClassName**

TextArea

**Purpose**

To obtain the number of columns in the TextArea.

**Syntax**

public int getColumns()

**Parameters**

None.

**Description**

This method returns the number of columns in the target TextArea object. This is the number of columns of the TextArea during its instantiation.

**Imports**

*import java.awt.TextArea;*

**Returns**

The number of columns of TextArea object; return type is int.

**Example**

Refer to Listing 5-1. In the action method of the textPanel class, the number of columns of the text area is used to create a new text area as a clone.

## getRows()

**ClassName**

TextArea

**Purpose**

To obtain the number of rows in the TextArea.

**Syntax**

public int getRows()

**Parameters**

None.

**Description**

This method returns the number of rows in the target TextArea object. This is the number of rows in the TextArea during its instantiation.

**Imports**

*import java.awt.TextArea;*

**Returns**

The number of rows of TextArea object; return type is int.

**Example**

Refer to Listing 5-1. In the action method of the class textPanel class, the number of rows of the text area that is used to create a new text area as a clone.

## insertText(*String, int*)

**ClassName**

TextArea

**Purpose**

Inserts the specified text at the specified index in the TextArea.

**Syntax**

public void insertText(String ins_text, int index)

**Parameters**

***ins_text***

The text to be inserted at specified index of the TextArea.

***index***

The index location in the existing text of TextArea where the new text is to be inserted.

**Description**

This method inserts the specified string at the specified index of the text contained in the target TextArea object. The value of the index should be less than the length of the already available text in the TextArea. If the index value exceeds the length of the existing text, Java will issue a StringIndexOutofBoundsException.

**Imports**

*import java.awt.TextArea;*

**Returns**

None.

**See Also**

The appendText method of the TextArea class

**Example**

Refer to Listing 5-1. In the action method of the textPanel class, name string n_str is inserted in the text area using this method.

## minimumSize(*int, int*), minimumSize()

**ClassName**

TextArea

**Purpose**

To obtain the minimum size dimension of the TextArea if no parameter is specified. If parameters are specified, this method obtains the minimum Dimensions for the specified number of rows and columns.

**Syntax**

public Dimension minimumSize(int rows, int cols)
public Dimension minimumSize()

**Parameters**

***rows***

The specified number of rows for which minimum size is to be found.

***cols***

The specified number of columns for which the minimum size is to be found.

**Description**

The height and width of window Dimensions are different from the number of rows and columns of the TextArea. If rows and columns are not specified, the rows and columns of the target TextArea object are taken as the values. The number of rows and columns indicate the number of characters accommodated within the space, whereas the Dimension indicates the window dimensions. For example, this method would be helpful in resizing a window or a frame containing a TextArea or in determining where to add the text area in the window.

**Imports**

*import java.awt.TextArea;*

**Returns**

The minimum Dimensions for a TextArea with the number of rows and columns. Return type is Dimension.

**See Also**

The preferredSize of the TextArea class; the Dimension class

**Example**

Refer to Listing 5-1. This method is used to print out the minimum width and height required when the CloneTextArea button is selected. This occurs in the action method of the textPanel class.

## paramString()

**ClassName**

TextArea

**Purpose**

To obtain the parameter String of the target TextArea object.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method obtains the String representation of the target TextArea object containing the parameters, rows, columns and text contained in it. This method is protected and hence can be used only by classes within the java.awt package. This method overrides the paramString method of class TextComponent.

**Imports**

*import java.awt.TextArea;*

**Returns**

The parameter string of type String.

**See Also**

The paramString method of the TextComponent class; the TextArea class

**Example**

The following code uses paramString by subclassing the TextArea.

```
package java.awt;
import java.awt.TextArea;

class myText extends TextArea {

       String myStringForm;

       public myText() {
        super("");
       }

        public String paramString() {
                        return super.paramString();
        }

        public static void main(String[] args) {
              myText txt = new myText();
              txt.getmyStringForm();
        }
}
```

## preferredSize(*int, int*), preferredSize()

**ClassName**

TextArea

**Purpose**

To obtain the preferred dimension of the TextArea if no parameter is specified. If parameters are specified, this method returns the preferred dimension for the specified rows and columns.

**Syntax**

public Dimension preferredSize(int rows, int cols)
public Dimension preferredSize()

**Parameters**

***rows***

The specified number of rows for which the preferred size is to be found.

***cols***

The specified number of columns for which the preferred size is to be found.

**Description**

The height and width of window dimensions are different from the number of rows and columns of the TextArea. If the rows and columns are not specified, the rows and columns of the target TextArea object are taken as the values. The number of rows and columns indicate the number of characters accommodated within the space, whereas the Dimension indicates the window dimensions. For example, this method would be helpful in resizing a window or frame containing a TextArea, or in determining where to position the textArea in the window. It returns the preferred size Dimensions for the text area.

**Imports**

*import java.awt.TextArea;*

**Returns**

The preferred Dimensions for a TextArea with the number of rows and columns. Return type is Dimension.

**See Also**

The minimumSize method in the TextArea class; the Dimension class

**Example**

Refer to Listing 5-1. This method is used to print out the preferred width and height required when the CloneTextAreabutton is selected. This occurs in the action method in the textPanel class.

## replaceText(*String, int, int*)

**ClassName**

TextArea

**Purpose**

The specified text replaces the existing text between the specified positions in the TextArea.

**Syntax**

plic void replaceText(String new_text, int start, int end)

**Parameters**

***new_text***

The specified text to replace the existing text between specified positions.

***start***

The beginning index location in the existing text of TextArea where the replacement text is to be inserted in place of the existing text.

***end***

The ending index location in the existing text of TextArea specifying the last point at which the existing text is to be replaced by the *new_text.*

**Description**

This method replaces the text between the specified locations, start and end, with the specified text, *new_text.* The length of the *new_text* need not be the same as that of the text being replaced; however, the values of the indexes should be less than the length of the already available text in the TextArea. If the index value exceeds the length of the existing text, an StringIndexOutofBoundsException is issued at runtime.

**Imports**

*import java.awt.TextArea;*

**Returns**

None.

**See Also**

The insertText method in the TextArea class

**Example**

Refer to Listing 5-1. In the action method of the textPanel class, replaceText method is used to change the "Enter Text Here" string to "Hello! "

## TextField

**Purpose**

A single line editor and a subclass of TextComponent.

**Syntax**

public class TextField extends TextComponent

**Description**

The TextField provides a single line for editing purposes and an interface for the user to enter text. It subclasses the TextComponent class with the default set to editable mode. It can be set to noneditable mode but, in most cases there would be no reason for you to do so. When RETURN is pressed in the TextField, an event ACTION_EVENT is posted. Appropriate event handling routines (e.g., the methods action() or handleEvent()) should be overridden to handle the events generated. Methods of TextComponent class allowing selection and manipulation of the selected text can be performed on an editable text field. Figure 5-10 illustrates the inheritance relationship of the TextField class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.TextField;*

**Constructors**

public TextField()
public TextField(int t_cols)

public TextField(String text)
public TextField(String text, int t_cols)

**Parameters**

***t_cols***

The number of columns specified in the TextField.

***text***

The text which forms the initial text contents of this TextField.

**Example**

An instance of the TextField class is a member in the textPanel class in Listing 5-1. The member object is name_f, signifying name field.

**Figure 5-10** Class diagram of the TextField class

## addNotify()

**ClassName**

TextField

**Purpose**

This method creates a peer of the target TextField object.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

An instance of the TextFieldPeer is created as a peer for the target TextField object. Using the peer, you can change the appearance of the TextArea without modifying its functionality. Required if you are writing your own AWT.

**Imports**

*import java.awt.TextField;*

**Returns**

None.

**See Also**

The TextFieldPeer class

**Example**

Refer to Chapter 9 describing the peers and interface for details.

## echoCharIsSet()

**ClassName**

TextField

**Purpose**

The boolean value indicating whether a character is set for echoing in the target TextField object.

**Syntax**

public boolean echoCharIsSet()

**Parameters**

None.

**Description**

In an object of type TextField, you can set an echo character associated with the field. Whenever you type in a character, only the echo character is displayed in the field and not the original characters you typed in. This is useful when you don't want anyone to see the characters you are entering (for example, in the case of a password or social security number). This method returns the boolean value of true if an echo character is set in the target TextField object; otherwise, it returns false.

**Imports**

*import java.awt.TextField;*

**Returns**

Boolean value of true if echo character is set; false otherwise.

**See Also**

The setEchoCharacter method of the TextField class

**Example**

Refer to Listing 5-1. In the constructor of the textPanel class, echo character of '*' is set, if it is not set earlier. This method is used to determine whether the echo character is already set.

## getColumns()

**ClassName**

TextField

**Purpose**

To obtain the number of columns in the TextField.

**Syntax**

public int getColumns()

**Parameters**

None.

**Description**

This method returns the number of columns in the target TextField object. This is the number of columns set for the TextField during its instantiation.

**Imports**

*import java.awt.TextField;*

**Returns**

The number of columns of TextField object; return type is int.

**Example**

Refer to Listing 5-1. This method is used to print the number of columns in the text field, name_f, in the action method in the textPanel class.

## getEchoChar()

**ClassName**

TextField

**Purpose**

To obtain the character used for echoing in the target TextField object.

**Syntax**

public char getEchoChar()

**Parameters**

None.

**Description**

In an object of type TextField, you can set an echo character associated with the field. Whenever you type in a character, only the echo character is displayed in the field and not the original characters you typed in. This is useful when you don't want anybody nearby to see the characters you are entering (for example, in the case of password or social security number). This method returns the character that has been set for echoing in the target TextField object, if it is set. If the echo character is not set, it returns a null character.

**Imports**

*import java.awt.TextField;*

**Returns**

The character set for echoing in the TextField. Return type is char.

**See Also**

The echoCharIsSet and setEchoCharacter methods of the TextField class

**Example**

Refer to Listing 5-1. In the constructor of class textPanel, echo character of '*' is set, if it is not set earlier. This method is used to print the echo character to the standard output.

## minimumSize(*int*), minimumSize()

**ClassName**

TextField

**Purpose**

Returns the minimum size dimension of the TextField if no parameter is specified. If a parameter is specified, then the minimum size dimension for the specified number of columns is returned.

**Syntax**

public Dimension minimumSize(int cols) public Dimension minimumSize()

**Parameters**

***cols***

The specified number of columns of the TextField for which the minimum size is to be found.

**Description**

The width of the window dimension is different from the number of columns in the TextField. The number of columns for which the minimum size is to be found can be specified. If it is not specified, the number of columns of the target TextField object is taken as the value. The number of columns indicates the number of characters that can be accommodated within the space whereas the Dimension indicates the window dimensions. For example, this method is helpful to resize a window or a frame containing a TextField or to determine where in the window to add the text field.

**Imports**

*import java.awt.TextField;*

**Returns**

The minimum Dimensions for a TextField with the number of columns. Return type is Dimension.

**See Also**

The preferredSize method of the TextField class; the Dimension class

**Example**

Refer to Listing 5-1. This method is used to print out the minimum width and height required when the ChangeText button is selected. This occurs in the action method in the textPanel class.

## paramString()

**ClassName**

TextField

**Purpose**

To obtain the parameter String of the target TextField object.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method returns the String representation of the target TextField object containing the parameters column and text contained in it. Because it is protected, this method can be used only by the classes within the *java.awt* package. This method overrides the paramString method in the TextComponent class.

**Imports**

*import java.awt.TextField;*

**Returns**

The parameter string of type String.

**See Also**

The TextField class; the paramString method of the TextComponent class

**Example**

Use of this method is similar to the example given for the paramString method in the TextArea class.

## preferredSize(*int*), preferredSize()

**ClassName**

TextField

**Purpose**

Returns a preferred size dimension for the TextField if no parameter is specified. If parameters are specified, then the preferred size dimensions for the specified columns is returned.

**Syntax**

public Dimension preferredSize(int cols)
public Dimension preferredSize()

**Parameters**

***cols***

The specified number of columns for which the preferred size is desired to be found.

**Description**

The width of window Dimensions is different from the number of columns in the TextField. The number of columns for which the preferred size is to be found can be specified. If it is not specified, the number of columns in the target TextField object is taken as the value. The number of columns indicates the number of characters that can be accommodated within the space, whereas the Dimension indicates the window dimensions. For example, this method is helpful to resize a window or a frame containing a TextField or to determine where in the window to add the text field. It returns the preferred size Dimensions for the text field.

**Imports**

*import java.awt.TextField;*

**Returns**

The preferred Dimensions for a TextField with the number of rows and columns. The return type is Dimension.

**See Also**

The minimumSize method of the TextField class; the Dimension class

**Example**

Refer to Listing 5-1. This method is used to print out the preferred width and height required when the ChangeText button is selected. This occurs in the action method in the textPanel class.

## setEchoCharacter(*char*)

**ClassName**

TextField

**Purpose**

Sets the specified character as the echo character for the target TextField object.

**Syntax**

public void setEchoCharacter(char echo_c)

**Parameters**

***echo_c***

The character to be echoed to the screen to represent any input to the text field.

**Description**

In an object of type *TextField,* you can set an echo character associated with the field. Whenever you type in a character, only the echo character is displayed in the field and not the original characters you typed in. This is useful when you don't want anyone to see the characters you are entering (for example, in the case of a password or social security number). This method sets the specific character for echoing in the target TextField object. After invocation of this method, a call to echoCharIsSet returns true.

**Imports**

*import java.awt.TextField;*

**Returns**

None.

**See Also**

The echoCharIsSet and getEchoChar methods of the TextField class

**Example**

Refer to Listing 5-1. In the constructor of the textPanel class, this method is used to set an echo character of '*', if it has not been set earlier.

## Dialog

**Purpose**

Creates a window that takes input from the user.

**Syntax**

public class Dialog extends Window

**Description**

The Dialog provides a pop-up window which takes input from the user. It must be bound to a Frame on construction. The default layout for a Dialog window is BorderLayout. The dialog window can be made modal, i.e., users are prevented from performing anything on other windows until they close the dialog. A dialog can also be non-modal. You can provide the option of resizing the window if needed. It can have a title and/or a border associated with it. Dialog subclasses the Window class. It helps in establishing communication between the user and the application. Figure 5-11 illustrates the inheritance relationship of the Dialog class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.Dialog;*

**Constructors**

public Dialog(Frame parent, boolean modal)
public Dialog(Frame parent, String title, boolean modal)

**Parameters**

***parent***

The parent frame to which the dialog is bound.

***title***

The title of the Dialog window.

***modal***

Boolean value indicating whether you want the Dialog to be modal or not.

**Example**

The dialogDemo class implemented in the following example, Listing 5-2 illustrates the use of Dialog and FileDialog classes and their member methods. The SaveDialog class subclasses the Dialog class and is used to indicate that the Save function is not implemented. Figure 5-12 shows the resultant dialogDemo window.

**Figure 5-11** Class diagram of the Dialog class

**Figure 5-12** The dialogDemo window

**Listing 5-2** dialogDemo.java: Progam demonstrating the use of methods in Dialog and FileDialog classes

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

/**
        file name: dialogDemo.java
        classes: dialogDemo
                 JavaFilenameFilter
                 QuitDialog
        Purpose: Illustration of classes Dialog and FileDialog
*/

public class dialogDemo extends Frame {

       MenuBar mbar;
           Menu f_menu; // to demostrate dialog and file dialog

           public dialogDemo() {

                      f_menu = new Menu("File");
                      f_menu.add("Open");
                      f_menu.add("Save");
                      f_menu.add("Quit");

               mbar = new MenuBar();
                    mbar.add(f_menu);
               setMenuBar(mbar);
                    pack();
                    show();
```

```
       }

   public boolean action(Event evt, Object arg) {

      if (evt.target instanceof MenuItem) {
         if (“Open".equals(arg)) {
           System.out.println(“ File Open called ”);

                          // usage of FileDialog
      FileDialog fd = new FileDialog(this, "File Window”);
      fd.setFilenameFilter(new JavaFilter());
                         // set the default directory to C: in windows
95
                         String dir = new String(“C:”);
                         // in case of Solaris set it to /opt
                         // String dir = new String(“/opt”);

                          fd.setDirectory(dir);
                          fd.setFile(“java”);
      fd.show();
      System.out.println(“ file is " + fd.getFile());
      System.out.println(“ Dir is " + fd.getDirectory());
      FilenameFilter f_f = fd.getFilenameFilter();
      System.out.print(“ mode is ”);
         if (fd.getMode()== FileDialog.LOAD)
              System.out.println(“ LOAD ”);
       else if (fd.getMode() == FileDialog.SAVE)
              System.out.println(“ SAVE ”);
      }
      if (“Save".equals(arg)) {
           // usage of dialog
           SaveDialog dial = new SaveDialog(this,"save not
implemented”);
           dial.pack();
      System.out.println(“ Title of the dialog is “ + dial
.getTitle());
           dial.resize(200,100);
           if (dial.isResizable())
                     System.out.println(“ Default is resizable”);
           dial.setResizable(false);
           dial.show();
   }
   if (“Quit".equals(arg)) {
                       System.exit(0);
   return true;
   }

         }
   return true;
 }

 public static void main(String args[]) {
                    dialogDemo dial = new dialogDemo();
                    dial.setTitle(“Dialogs Demo”);

                    dial.pack();
                    dial.show();
```

```
   }
}

// SaveDialog subclasses Dialog class

class SaveDialog extends Dialog {

        String str;
        public SaveDialog(Frame parent, String s){

        super(parent,false);
        str = s;
         setBackground(Color.gray);
          setLayout(new BorderLayout());
        setTitle("Save Dialog");
         Panel p = new Panel();
         p.add(new Button("OK"));
         add("South",p);
        System.out.print(" I am ");
        if (!this.isModal())
                System.out.println(" NOT ");
        System.out.println(" modal");

        }

        public boolean action(Event evt, Object arg) {

            if ("OK".equals(arg)) {
                dispose();
                return true;
            }
            return false;
   }

   public void paint(Graphics g){
      g.setColor(Color.white);
             g.drawString(str, 50,20);
   }
}

// class implementing the FilenameFilter interface

class JavaFilter implements FilenameFilter {

  String suffix;
  public JavaFilter() {
   suffix = ".java";
   System.out.println(" JavaFilter created ");
  }

  public boolean accept(File dir, String name) {
   String file = dir.getName();
   if ((file.substring(file.length() - 5,
file.length())).equals(suffix)){

      System.out.println(" suffix equals java ");
      return true;
```

```
   } else {
      System.out.println(" suffix not .java");
      return false;
   }
  }
}
```

## addNotify()

**ClassName**
Dialog
**Purpose**
This method creates a peer of the target Dialog object.
**Syntax**
public synchronized void addNotify()
**Parameters**
None.
**Description**
An instance of the DialogPeer is created as a peer for the target Dialog object. Using the peer you can change the appearance of the Dialog window without modifying its functionality. Required if you are writing your own AWT.
**Imports**
*import java.awt.Dialog;*
**Returns**
None.
**See Also**
The DialogPeer class
**Example**
Refer to Chapter 9 describing the peers and interface for details.

## getTitle()

**ClassName**
Dialog
**Purpose**
Obtains the title of the Dialog window, if it has already been set.
**Syntax**
public String getTitle()
**Parameters**
None.
**Description**
A Dialog window can be identified by its title. This title appears at the top frame border of the dialog window. A Dialog can also be constructed without any title. This method returns the title of the target Dialog object if it has been set.
**Imports**

*import java.awt.Dialog;*

**Returns**

The title of the Dialog window, if set earlier. The return type is String.

**See Also**

The setTitle method of the Dialog class

**Example**

Refer to Listing 5-2. The title of the SaveDialog object is printed to the screen using this method.

## isModal()

**ClassName**

Dialog

**Purpose**

The boolean value indicating whether or not the target Dialog window is modal is returned.

**Syntax**

public boolean isModal()

**Parameters**

None.

**Description**

A Dialog window can be modal. This is specified in its constructor. If a dialog window is modal, the user is prevented from performing any action on the parent frame when the Dialog window pops up. The user can work on the parent frame only after the dialog window is closed. This is helpful to convey occurrence of fatal errors in the application or when input is necessary from the user to perform the next step in the application. This method returns true if the Dialog window is modal; false if the window is not modal.

**Imports**

*import java.awt.Dialog;*

**Returns**

Boolean value of true is returned if the target Dialog object is modal; value of false is returned if the target Dialog object is non-modal.

**See Also**

Constructors of the Dialog class

**Example**

Refer to Listing 5-2. This method is used to print whether or not the SaveDialog is modal.

## isResizable()

**ClassName**

Dialog

**Purpose**

Returns the boolean value indicating whether the target Dialog window is resizable.

**Syntax**

public boolean isResizable()

**Parameters**

None.

**Description**

A Dialog window can be made resizable by the user. Using the setResizable method of the Dialog class, a Dialog object is made resizable or nonresizable. The default is resizable. This method returns true if the target Dialog object is resizable; false otherwise.

**Imports**

*import java.awt.Dialog;*

**Returns**

Boolean value of true is returned if the target Dialog object is resizable; false if the target Dialog object is not resizable.

**See Also**

The setResizable method of the Dialog class

**Example**

Refer to the example program in Listing 5-2. The SaveDialog object is verified as being resizeable by default using this method.

## paramString()

**ClassName**

Dialog

**Purpose**

Obtains the parameter String of the target Dialog object.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

Returns the String representation of the target Dialog object containing the parameters Frame, title, and value indicating whether it is modal. This method is protected and hence, can be used only by classes within the *java.awt* package. This method overrides the paramString method of the Container class.

**Imports**

*import java.awt.Dialog;*

**Returns**

The parameter string of type String.

**See Also**

The paramString method of the Container class; the Dialog class

**Example**

Use this method similar to the example given for the paramString method of the TextArea class.

## setResizable(*boolean*)

**ClassName**

Dialog

**Purpose**

Allows the boolean value indicating whether the target Dialog window is resizable to be set.

**Syntax**

public void setResizable(boolean ok2resize)

**Parameters**

***ok2resize***

The boolean value indicating whether you want the Dialog window to be resizable. True indicates that the Dialog window should be resizable. False indicates that the Dialog window should not be resizable.

**Description**

A Dialog window can be made resizable or prevented from being resized by the user by using this method. The default is resizable. The boolean value indicating whether or not a Dialog window is resizable can be found using the isResizable method in the Dialog class.

**Imports**

*import java.awt.Dialog;*

**Returns**

None.

**See Also**

The isResizable method of the Dialog class

**Example**

Refer to the example in Listing 5-2. In the action method in the dialogDemo class, the SaveDialog is set to be nonresizable using this method.

## setTitle(*String*)

**ClassName**

Dialog

**Purpose**

The title of the Dialog window is set to the specified string value using this method.

**Syntax**

public void setTitle(String title)

**Parameters**

***title***

A string value which you want to be set as the title of the target Dialog object.

**Description**

A Dialog window can be identified by its title. This title appears at the top frame border of the dialog window. A Dialog can be constructed without any title also. This method can be used to set a title, if one has been set during construction, or to rename a Dialog window. The getTitle method in the Dialog class is used to obtain the title if it has been set.

**Imports**

*import java.awt.Dialog;*

**Returns**

None.

**See Also**

The getTitle method of the Dialog class

**Example**

Refer to the example program in Listing 5-2. The title of the SaveDialog is set to "Save Dialog" using this method in the constructor of the class SaveDialog.

## FileDialog

**Purpose**

A modal Dialog window displaying a file selection dialog.

**Syntax**

public class FileDialog extends Dialog

**Description**

The FileDialog provides a pop-up window which helps the user select a file. FileDialog window is a modal window; hence, the user cannot perform any action on other windows until this FileDialog window is closed. The window is disposed of once the user selects a file or cancels the selection. A FileDialog window has to be bound to a frame parent. The frame parent is found if this FileDialog object is declared a member of a subclass of Frame. Alternatively, you can look back in the tree hierarchy until you find a parent of type Frame. A FileDialog can be constructed in either LOAD or SAVE mode. This characteristic can be specified during construction. The public variable members of this class, LOAD and SAVE, are used to set the mode. After the user closes the FileDialog window by selecting a file, the selection is available within the object instance and can be obtained by using the getFile method of class FileDialog. Figure 5-13 illustrates the inheritance relationship of class FileDialog.

**PackageName**

*java.awt*

**Imports**

*import java.awt.FileDialog;*

**Constructors**

public FileDialog(Frame parent, String title)
public FileDialog(Frame parent, String title, int mode)

**Parameters**

***parent***

The parent frame to which the FileDialog is bound.

***title***

The title of the FileDialog window.

***mode***

Boolean value indicating whether the FileDialog should be in SAVE or LOAD mode.

**Variables**

public final static int LOAD—used to set the FileDialog to load mode as the selected file is to be loaded.

public final static int SAVE—used to set the FileDialog to save mode as the selected file is to be saved.

**Example**

The dialogDemo class in Listing 5-2 uses the FileDialog class to pop up a file dialog window for selecting a file using this class. This happens when Open is selected from the File menu in the UI generated by the listing.

**Figure 5-13** Class diagram of the FileDialog class

## addNotify()

**ClassName**

FileDialog

**Purpose**

This method creates a peer of the target FileDialog object.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

An instance of the FileDialogPeer is created as a peer for the target FileDialog object. Using the peer, you can change the appearance of the FileDialog window without modifying its functionality. This method is required if you are writing your own AWT. It overrides the addNotify method in the Dialog class.

**Imports**

*import java.awt.FileDialog;*

**Returns**

None.

**See Also**

The FileDialogPeer clas

**Example**

Refer to Chapter 9 describing the peers and interface for details.

## getDirectory()

**ClassName**

FileDialog

**Purpose**

Obtains the directory of the FileDialog.

**Syntax**

public String getDirectory()

**Parameters**

None.

**Description**

This method gets the directory of the file dialog window. A FileDialog has a directory, a file, and a filename filter associated with it. The default directory is the directory where you execute the program. You can change the initial directory of the FileDialog by using the setDirectory method.

**Imports**

*import java.awt.FileDialog;*

**Returns**

The directory of the FileDialog window is returned. Return type is String.

**See Also**

The setDirectory method of the FileDialog class

**Example**

Refer to the example program in Listing 5-2. The directory selected is printed to the screen after the user closes the file dialog window that pops up when Open is selected in the File menu.

## getFile()

**ClassName**

FileDialog

**Purpose**

Obtains the name of the selected file using the FileDialog.

**Syntax**

public String getFile()

**Parameters**

None.

**Description**

This method gets the selected file name using the file dialog window. A FileDialog has a directory, a file, and a filename filter associated with it. By selecting a series of the components from the file dialog window, a user finally selects a file or cancels the selection operation. After the window is closed, this method can be used to obtain the name of the selected file. The String returned by this method denotes the file name. If the user has canceled the selection, this method returns a null String.

**Imports**

*import java.awt.FileDialog;*

**Returns**

The name of the file selected by the user using the FileDialog window, or null if the selection has been canceled. Return type is String.

**See Also**

The setFile method of the FileDialog class

**Example**

Refer to the example program in Listing 5-2. The selected file name is printed to the screen after the user closes the file dialog window that pops up when Open is selected in the File menu.

## getFilenameFilter()

**ClassName**

FileDialog

**Purpose**

Obtains the selected filter using the FileDialog.

**Syntax**

public FilenameFilter getFilenameFilter()

**Parameters**

None.

**Description**

This method gets the filename filter of the FileDialog window. A FileDialog has a directory, a file, and a filename filter associated with it. You can implement the FilenameFilter interface in your class and set the filter of a FileDialog object to it. Invoking this method on a FileDialog object returns the filter to which the FileDialog is set.

**Imports**

*import java.awt.FileDialog;*

**Returns**

The filename filter of the FileDialog window. Return type is FilenameFilter.

**See Also**

The setFilenameFilter method of the FileDialog class; the java.io.FilenameFilter interface

**Example**

Refer to the example program in Listing 5-2. The string form of FilenameFilter that is set in the file dialog, is printed to the screen after the user closes the file dialog window which pops up when Open is selected in the File menu.

## getMode()

**ClassName**

FileDialog

**Purpose**

Returns the mode of the FileDialog window. It can be either LOAD or SAVE mode.

**Syntax**

“public int getMode()

**Parameters**

None.

**Description**

A FileDialog can be constructed in either LOAD or SAVE mode. This characteristic can be specified during construction. The public variable members of this class, LOAD and SAVE, are used to compare the mode. This method is used to determine the mode in which a FileDialog window was opened.

**Imports**

*import java.awt.FileDialog;*

**Returns**

The mode in which the target FileDialog object was created. Return type is int and that value can be compared with the LOAD and SAVE variables of FileDialog class.

**See Also**

Constructors in the FileDialog class

**Example**

Refer to the example program in Listing 5-2. The mode in which the file dialog window was opened is printed to the screen after the user closes the file dialog window that pops up when Open is selected in the File menu.

## paramString()

**ClassName**

FileDialog

**Purpose**

Obtains the parameter String of the target FileDialog object.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method returns the String representation of the target FileDialog object containing the parameters file, mode, and directory. This method is protected and hence, can be used only by the classes within the *java.awt* package. This method overrides the paramString method in the Dialog class.

**Imports**

*import java.awt.FileDialog;*

**Returns**

The parameter string of type String.

**See Also**

The paramString method of the Dialog class; the FileDialog class

**Example**

Use this method similar to the example given for the paramString method in the TextArea class.

## setDirectory(*String*)

**ClassName**

FileDialog

**Purpose**

Sets the specified directory of the FileDialog.

**Syntax**

public void setDirectory(String dir)

**Parameters**

***dir***

Directory name to be set for the FileDialog window.

**Description**

This method sets the directory of the file dialog window to the specified directory name. A FileDialog has a directory, a file, and a filename filter associated with it. The directory of a FileDialog object can be obtained using the getDirectory method.

**Imports**

*import java.awt.FileDialog;*

**Returns**

None.

**See Also**

The getDirectory method of the FileDialog class

**Example**

Refer to the example program in Listing 5-2. The directory is set to "C:" in Windows 95. It is set to /opt (to be uncommented in code) in Solaris.

## setFile()

**ClassName**

FileDialog

**Purpose**

Sets the file for the target FileDialog object to the specified file name.

**Syntax**

public void setFile(String file)

**Parameters**

***file***

Name of the specified file, which is to be set as the file for the dialog window.

**Description**

This method sets the specified file name for the target FileDialog window. A FileDialog has a directory, a file, and a filename filter associated with it. A user, by a series of selections from the components of the file dialog window, finally selects a file or cancels the selection operation. After the window is closed, the selected filename is available as a member of the file dialog window and can be accessed using the getFile method.

**Imports**

*import java.awt.FileDialog;*

**Returns**

None.

**See Also**

The getFile method of the FileDialog class

**Example**

Refer to the example program in Listing 5-2. The file is set to “java” in the action method if Open is selected from the menu.

## setFilenameFilter(FilenameFilter)

**ClassName**

FileDialog

**Purpose**

Sets the specified filter for the target FileDialog object.

**Syntax**

public void setFilenameFilter(FilenameFilter filter)

**Parameters**

***filter***

The specified FilenameFilter to be set to the target FileDialog window.

**Description**

This method sets the filename filter of the FileDialog window. A FileDialog has a directory, a file, and a filename filter associated with it. The FileNameFilter object assists in masking the directory with the specified string mask. Only files that are accepted by the filter will form the available selection among all files in that directory.

**Imports**

*import java.awt.FileDialog;*

**Returns**

None.

**See Also**

The getFilenameFilter method of the FileDialog class

**Example**

Refer to the example program in Listing 5-2. FilenameFilter is set to an instance of JavaFilter.

## List

**Purpose**

A component that provides a scrolling list of text items from which the user can select one or many items.

**Syntax**

public class List extends Component

**Description**

List provides a selection mechanism that allows users to select from an unlimited number of choices. The choices are provided as a scrolling list of items. Selection among limited numbers of choices is provided using Menus, Choice, and Checkboxes which are discussed in Chapter 6. A List can be set such that it allows either a single or multiple selection. An event gets posted on the selection of item(s) and appropriate event handling routines are used to take the necessary action. You have to write your own event handler to handle the LIST_SELECT

and LIST_DESELECT events that are posted when items are selected or deselected, respectively. Figure 5-14 illustrates the inheritance relationship of the List class.

**PackageName**

*java.awt*

**Imports**

*import java.awt.List;*

**Constructors**

public List()
public List(int visible_rows, boolean multipleOk)

**Parameters**

***visible_rows***

The number of rows of visible lines in the List.

***multipleOk***

Boolean value that specifies whether multiple selection is allowed.

**Example**

The viewList class implemented in the following example, Listing 5-3, uses the class List to demonstrate the methods in List. Figure 5-15 shows the window that results.

**Figure 5-14** Class diagram of the List class


**Figure 5-15** viewList window created by executing Listing 5-3

**Listing 5-3** viewList.java: Demonstrating the usage of methods of the List class

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

/**
        Filename: viewList.java
        classes:    viewList
        Purpose: demonstrating the usage of List class and its methods
*/

class viewList extends Frame {
  List list;
  Button cls;
  int selected;
  TextField txt_f;
```

```
public viewList() {

   list = new List(3,false);
                        // three rows; multiple selection disabled
   add("North",list);
   setTitle("List Demo");
   Panel f_p = new Panel();
   f_p.setLayout(new FlowLayout());
   cls = new Button("Clear");
   f_p.add(cls);
   f_p.add(new Button("Select2"));
   f_p.add(new Button("Delete"));
   add("Center",f_p);
   Panel t_p = new Panel();
   txt_f = new TextField("Standby",10);
   t_p.add(txt_f);
   t_p.add(new Button("Replace"));
   add("South",t_p);
   list.addItem("One");
   list.addItem("Two");
   list.addItem("Three");
   list.addItem("Five");
   list.addItem("Four",3);
   int count = list.countItems();
   System.out.println("count is " + count);
   list.addItem("Six",count);
   if (!list.allowsMultipleSelections()) {
        list.setMultipleSelections(true);
   }

   selected = 0;
   pack();
   show();
   System.out.println(" number of visible items is "
        + list.getRows());
   System.out.println("Min size of list "
        + list.minimumSize().height);
   System.out.println("Min size with 20 rows: "
        + list.minimumSize(20).height);
   System.out.println("Preferred size of list "
        + list.preferredSize().height);
   System.out.println("Preferred size with 20 rows: "
        + list.preferredSize(20).height);
 }

   public boolean handleEvent(Event evt) {

       if (evt.id == Event.LIST_SELECT) {
       selected++;
       System.out.println(" List items selected are:");
       String[] items = list.getSelectedItems();
       int[]   indexes = list.getSelectedIndexes();
        System.out.println(" number of visible items is "
            + list.getRows());
       try {
         for (int i =0;items[i]!=null;i++) {
```

```
          System.out.print(items[i]);
       if (list.isSelected(indexes[i])) {
         System.out.println(" index is " + indexes[i]);
         }
       System.out.println(" visible index is "
         + list.getVisibleIndex());
       if (list.getVisibleIndex()!=0)
        list.makeVisible(0);

   }
} catch (ArrayIndexOutOfBoundsException ae) {}
 // try selecting fourth item at index 3
     if (list.isSelected(3)) {
      list.deselect(3);
      selected--;
      System.out.println(" Sorry! Item temporarily
          not available");
        }
      }
     if (evt.id == Event.LIST_DESELECT) {
      selected--;
      System.out.println(" List items selected are:");
      String[] items = list.getSelectedItems();
      int i=0;
      try {
      while (items[i]!=null) {
      System.out.println(items[i++]);
     }
     } catch (ArrayIndexOutOfBoundsException ae) {}

   }
   if (evt.id == Event.ACTION_EVENT){

      if (evt.target instanceof Button) {
       Button sel = (Button)evt.target;
   if (sel.getLabel().equals("Clear")) {
       list.clear();
     selected =0;
 }
      if (sel.getLabel().equals("Delete")) {
         selected--;
          System.out.println(list.getSelectedItem() +
               " is deleted");
          list.delItem(list.getSelectedIndex());
                      // list.delItems(1,3);
      }
      if (sel.getLabel().equals("Select2")) {
         list.select(1);
       if (list.allowsMultipleSelections())
             selected++;
      else selected = 1;
         System.out.println(" List items selected are:");

         String[] items = list.getSelectedItems();
         // System.out.println(list.getSelectedItem());
            int i=0;
            try {
```

```
              while (i<selected) {
                 System.out.println(items[i++]);
              }
        } catch (ArrayIndexOutOfBoundsException ae) {
        System.out.println(" exception");}
    }
    if (sel.getLabel().equals("Replace")) {
       int tmp = list.getSelectedIndex();
       System.out.println(list.getItem(tmp) +
           "is replaced now ");
       list.replaceItem(txt_f.getText(),tmp);
      }

   }
  }
   return true;

}

       public static void main(String args[]) {
           viewList vl = new viewList();
           vl.setTitle("List Demo");
           vl.pack();
           vl.show();
    }
}
```

## addItem(*String*), addItem(*String, int*)

**ClassName**

List

**Purpose**

An item is added to the scrolling list. If an index is specified, the item is added at that index of the list; otherwise, it is added to the end of the scrolling list.

**Syntax**

public synchronized void addItem(String item)
public synchronized void addItem(String item, int index)

**Parameters**

***item***

The String representing the item to be added to the List.

***index***

The position in the list at which to insert the specified item.

**Description**

This method adds an item to the list of options in this List. The item gets added to the end of the list if the index is not specified. While specifying the index you should note that the index starts at zero. So the first item has an index value of zero, second item has index value of one, and so on. This method is a synchronized method, so at a given instant only one Thread can add an item when multiple Threads are trying to add items to the target List object.

**Import**

*import java.awt.List;*

**Returns**

None.

**See Also**

The delItem(int) and delItems(int, int) methods of the List class

**Example**

Refer to the Listing 5-3. Items are added using this method in the constructor.

## addNotify()

**ClassName**

List

**Purpose**

This method creates a peer of the target List object.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

An instance of the ListPeer is created as a peer for the target List object. Using the peer, you can change the appearance of the List without modifying its functionality. This method is required if you are writing your own AWT.

**Imports**

*import java.awt.List;*

**Returns**

None.

**See Also**

The ListPeer class

**Example**

Refer to Chapter 9 describing the peers and interface for details.

## allowsMultipleSelections()

**ClassName**

List

**Purpose**

Determines whether multiple selection is allowed in the target List object.

**Syntax**

public boolean allowsMultipleSelections()

**Parameters**

None.

**Description**

You can specify a List to allow multiple selections or only single selectionaccording to the usage of the List object. This method checks tosee if

thetarget List object allows multiple selection. You can specify whether you want this List to allow multiple selection by using the setMultipleSelections method.

**Imports**

*import java.awt.List;*

**Returns**

If the target List object allows multiple selection, this method returns true. It returns false if the List does not allow multiple selections.

**See Also**

The setMultipleSelections method of the List class

**Example**

Refer to Listing 5-3. This method is used to allow multiple selections in the constrcutor, if that has not been allowed earlier. The verification is done using this method.

## clear()

**ClassName**

List

**Purpose**

This method clears the list, i.e., it deletes all the items in the List.

**Syntax**

public void clear()

**Parameters**

None.

**Description**

All the items in a List object can be removed by making a single call to this method.

**Imports**

*import java.awt.List;*

**Returns**

None.

**See Also**

The delItem and delItems methods

**Example**

Refer to Listing 5-3. When the Clear button is pressed, the event handler clears the list using this method.

## countItems()

**ClassName**

List

**Purpose**

Obtains the number of items in the List.

**Syntax**

public int countItems()

**Parameters**

None.

**Description**

The method counts the number of items in the List. The count is automatically increased when a new item is added. It is also the maximum index in this List.

**Imports**

*import java.awt.List;*

**Returns**

The number of items in the List at a given instance. Return type is int.

**See Also**

The getItem() method of the List class

**Example**

Refer to Listing 5-3. The number of items in the list is printed to the screen using this method.

## delItem(*int*), delItems(*int, int*)

**ClassName**

List

**Purpose**

Remove an item at the specified index from the List, or remove multiple items between the specified start and end positions from the scrolling list.

**Syntax**

public synchronized void delItem(int index)
public synchronized void delItems(int start, int end)

**Parameters**

***index***

The index position, in the List, of the item to be deleted.

***start***

The index of the first item in the selected sequence of items to be deleted.

***end***

The index of the last item in the selected sequence of items to be deleted.

**Description**

This method deletes an item at the specified index, if an index is specified. If both the start and end indexes are specified, all the items between the given indexes are deleted from the list. The value of start index should not be more than the end index. When specifying the index you should note that the index starts at zero. So the first item has an index value of zero, the second item has an index value of one, and so on. This method is a synchronized method. At any given instance, only one Thread can delete item(s) from the List when multiple Threads are trying to access the target List object.

**Imports**

*import java.awt.List;*

**Returns**

None

**See Also**

The addItem(String) and addItems(String, int) methods of the List class

**Example**

Refer to Listing 5-3. When the Delete button is pressed after selecting an item in the list, this method is used to delete the item.

## deselect(*int*)

**ClassName**

List

**Purpose**

Deselects the item at the specified index in the List.

**Syntax**

public void deselect(int index)

**Parameters**

***index***

The index of the item to be deselected in the target List object.

**Description**

The method deselects an item that has been selected earlier. The index of the item is specified and the value of the index should not be more than the number of items in the list. If it is so, an IndexOutofBoundsException is issued at runtime. If you try to deselect a item which is not selected earlier, then there is no effect from this method.

**Imports**

*import java.awt.List;*

**Returns**

None.

**See Also**

The select(int) method of the List class

**Example**

Refer to Listing 5-3. Try selecting the fourth item in the list and observe what happens. Selection of the fourth item results in that item getting deselected immediately.

## getItem(*int*)

**ClassName**

List

**Purpose**

Returns the item at the specified index in the List.

**Syntax**

public String getItem(int index)

**Parameters**

***index***

The index of the item to be retrieved from the target List object.

**Description**

The method gets the item at the specified index in the target List object. The index starts from 0. Hence, the index of the first item is 0, the second item is 1, and so

on. If the specified index is greater than or equal to the number of items, ArrayOutOfBounds exception is issued at runtime.

**Imports**

*import java.awt.List;*

**Returns**

The item at the specified location in its String form is returned.

**See Also**

The countItems() method of the List class

**Example**

Refer to Listing 5-3. When Replace is pressed after selecting an item, this method is used to print the item replaced.

## getRows()

**ClassName**

List

**Purpose**

Obtains the number of visible lines in the target List object.

**Syntax**

public int getRows()

**Parameters**

None.

**Description**

This method gets the number of lines visible in the target List object.

**Imports**

*import java.awt.List;*

**Returns**

The number of rows visible in the List. Return type is int.

**Example**

Refer to Listing 5-3. Number of visible items in the list is determined inside the constructor using this method.

## getSelectedIndex(), getSelectedIndexes()

**ClassName**

List

**Purpose**

The selected index is returned in the case of single selection. An array of selected index values are returned in the case of multiple selection.

**Syntax**

public synchronized int getSelectedIndex()
public synchronized int[] getSelectedIndexes()

**Parameters**

None.

**Description**

The method gets the index(es) of the selected item(s). Invocation of the getSelectedIndex() method, after selecting multiple items, will result in an ArrayIndexOutofBoundsException being issued. Invoking the getSelectedIndexes() method on a single selection List will retrieve the single selected item. If there is no element selected, both methods will return a value of -1.

**Imports**

*import java.awt.List;*

**Returns**

The getSelectedIndex() method returns the index of the selected iteminasingle selection List object. The return type is int. The getSelectedIndexes() method returns an integer array containing the setofselected indexes.

**See Also**

The select, deselect, and isSelected methods of the List class

**Example**

Refer to Listing 5-3. The getSelectedIndexes method is used in the program as it is a multiple selection list. It is used to print the list of selected items.

## getSelectedItem(), getSelectedItems()

**ClassName**

List

**Purpose**

In case of single selection, returns the selected item. In the case of multiple selections, it returns an array of selected items.

**Syntax**

public synchronized String getSelectedItem()
public synchronized String[] getSelectedItems()

**Parameters**

None.

**Description**

The method gets the selected item(s). Invocation of getSelectedItem() method on multiple selection List will effect in a runtime exception to be issued. Invoking the getSelectedItems() method on a single selection List will return the selected item. If there is no element selected, both methods will return null.

**Imports**

*import java.awt.List;*

**Returns**

The getSelectedItem() method returns the String form of the selection item in a single selection List object. The return type is String. The getSelectedItems() method returns a String array containing the set of selected items.

**See Also**

The select, deselect, and isSelected methods of the List class

**Example**

Refer to Listing 5-3. The getSelectedItems method is used in the program, as it is a multiple selection list. It is used to print the list of selected items.

## getVisibleIndex()

**ClassName**

List

**Purpose**

This method returns the index that was last made visible by the makeVisible method in the List class. Default is -1.

**Syntax**

public int getVisibleIndex()

**Parameters**

None.

**Description**

The method gets the index of the item that was last forcibly made visible in the List. Though a List object allows users to select from an unlimited number of items, it can display only a part of them. Users have to scroll up and down to view other items. The makeVisible method is used to force a previously invisible item visible. This method returns the index of the item thus made visible. If makeVisible has not been called earlier, invoking this method will return -1.

**Imports**

*import java.awt.List;*

**Returns**

Index of the last item that was forcibly made visible using the makeVisible method.

**See Also**

The makeVisible method of the List class

**Example**

Refer to Listing 5-3. The visible index is printed out during LIST_SELECT event occurrence.

## isSelected(int)

**ClassName**

List

**Purpose**

The method checks to see if the item at the specified index is selected.

**Syntax**

public boolean isSelected(int index)

**Parameters**

***index***

The index of the item that is verified if selected.

**Description**

The method checks to see if the item at the specified index is selected. If it is selected, this method returns true. If the item is not selected, the method returns

false. The index starts from 0. Hence, the index of the first item is 0, the second item is 1, and so on. If the specified index is greater than or equal to the number of items, an ArrayOutOfBoundsException is issued at runtime.

**Imports**

*import java.awt.List;*

**Returns**

A boolean value indicating whether the item at the specified index has been selected.

**See Also**

The select and deselect methods of List class

**Example**

Refer to Listing 5-3. This method is used to determine if the item at index 3 (the fourth item) is selected and if so, it deselects it.

## makeVisible(*int*)

**ClassName**

List

**Purpose**

The item at the specified index is forcibly made visible in the List.

**Syntax**

public void makeVisible(int index)

**Parameters**

***index***

The index of the item to be made visible in the target List object.

**Description**

The method forces the item at the specified index in the target List object to be visible. The index starts at 0. Hence, the index of the first item is 0, the second item is 1, and so on. If the specified index is greater than or equal to the number of items, an ArrayOutOfBoundsException is issued at runtime.

**Imports**

*import java.awt.List;*

**Returns**

None.

**See Also**

The getVisibleIndex() method of the List class

**Example**

Refer to Listing 5-3. Whatever item you select from the list, the list is reset to be visible from the first item by using this method.

## minimumSize(*int*), minimumSize()

**ClassName**

List

**Purpose**

Obtains the minimum size Dimension of the target List object if no parameter is specified. If a parameter is specified, then the minimum size Dimensions for the specified number of rows is returned.

**Syntax**

public Dimension minimumSize(int rows)
public Dimension minimumSize()

**Parameters**

***rows***

The specified number of rows for which minimum size is to be found.

**Description**

The height and width of the window dimension are different from the number of rows of a List. If the parameter is not specified, the number of rows of the target List object is taken as the value. The number of rows indicates the number of items to be accommodated within the space, whereas the Dimension indicates the window dimensions. For example, this method is helpful to resize a window or a frame containing a List or to determine where in the window to add the List component.

**Imports**

*import java.awt.List;*

**Returns**

The minimum Dimensions for a List with the number of rows. Return type is Dimension.

**See Also**

The preferredSize method of the List class

**Example**

Refer to Listing 5-3. The minimum size details are obtained and printed in the constructor using these methods.

## paramString()

**ClassName**

List

**Purpose**

Obtains the parameter String of the target List object.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

Obtains the String representation of the target List object containing the selected item. This method is protected and hence, can be used only by classes within the *java.awt* package. This method overrides the paramString method of the Component class.

**Imports**

*import java.awt.List;*

**Returns**

The parameter string of type String.

**See Also**

The paramString method of the Component class,; the List class

**Example**

The following code uses paramString by subclassing the List class.

```
package java.awt;
import java.awt.List;

class myList extends List {
        String myStringForm;
        public myList() {
        super();
       }
        public String paramString() {
                        return super.paramString();
        }
        public static void main(String[] args) {
              myList lst = new myList();
              lst.getmyStringForm();
        }
}
```

## preferredSize(*int*), preferredSize()

**ClassName**

List

**Purpose**

Obtains the preferred size Dimension of the List if no parameter is specified. If a parameter is specified, the preferred size Dimension for the specified number of rows is returned.

**Syntax**

public Dimension preferredSize(int rows)
public Dimension preferredSize()

**Parameters**

***rows***

The specified number of rows for which the preferred size is desired to be found.

**Description**

The width of window dimension is different from the number of rows in the List. The number of rows for which the preferred size is to be found is specified. If it is not specified, the number of rows in the target List object is taken as the value. The number of rows indicates the number of items of the List that are visible at a time whereas the Dimension indicates the window dimensions. For example, this method is helpful to resize a window or a frame containing a List or to determine where, in the window, to add the list to. It returns the preferred size Dimensions for the List.

**Imports**

*import java.awt.List;*

**Returns**

The preferred Dimensions for a List with the number of rows. Return type is Dimension.

**See Also**

The minimumSize method of the List class; the Dimension class

**Example**

Refer to Listing 5-3. The preferred size details are obtained and printed in the constructor using these methods.

## removeNotify()

**ClassName**

List

**Purpose**

Removes the peer of this List object.

**Syntax**

public void removeNotify()

**Parameters**

None.

**Description**

A List peer is used to change the appearance of your list, without changing its functionality. This method removes the peer of this component.

**Imports**

*import java.awt.List;*

**Returns**

None.

**See Also**

The ListPeer class

**Example**

Refer to Chapter 9 describing Peer interfaces.

## replaceItem(*String, int*)

**ClassName**

List

**Purpose**

The specified item replaces the existing item at the specified index in the List.

**Syntax**

public void replaceItem(String new_item, int index)

**Parameters**

***new_item***

The specified item to replace the existing item at the specified position.

**index**

The index location of the existing item of List where the replacement item is to be inserted instead of the existing item.

**Description**

This method replaces the item at the specified index with that specified item, new_item. The values of the indexes should be less than the number of available items in the List. If the index value exceeds the number of items, an IndexOutofBoundsException is issued at Java runtime.

**Imports**

*import java.awt.List;*

**Returns**

None.

**See Also**

The addItemmethod of the List class

**Example**

Refer to Listing 5-3. This method is used to replace a selected item when the Replace button is pressed after selecting an item.

## select(*int*)

**ClassName**

List

**Purpose**

Selects the item at the specified index in the List.

**Syntax**

public void select(int index)

**Parameters**

***index***

The index of the item to be selected in the target List object.

**Description**

The method selects an item at the specified index. The value of the index should not be more than the number of items in the list. If it is so, an IndexOutofBoundsException is issued at runtime.

**Imports**

*import java.awt.List;*

**Returns**

None.

**See Also**

The deselect(int) method of the List class

**Example**

Refer to Listing 5-3. When the Select2 button is pressed, this method is used to select the item at location 2 in the list.

## setMultipleSelections(boolean)

**ClassName**

List

**Purpose**

Makes the necessary changes, depending on the passed boolean value, to the target List object to either allow or disallow multiple selections.

**Syntax**

public void setMultipleSelections(boolean multipleOk)

**Parameters**

***multipleOk***

If the value of this parameter is true, the List object allows multiple selections; if false, it allows only single selection.

**Description**

This method specifies a List to allow multiple selections or only a single selection. You can check whether the target List object allows multiple selections by using the allowsMultipleSelection method.

**Imports**

*import java.awt.List;*

**Returns**

None.

**See Also**

The allowsMultipleSelections method of the List class

**Example**

Refer to Listing 5-3. In the constructor of viewList, this method sets the list to multiple selection if it has not been set before.

## API Reference Interface Application

The application for this chapter is a basic framework for an API Reference Interface Application. Using the interface provided in this Java application, users can specify a class name and obtain information about any selection of methods in the specified class. They can view the details of any method and optionally can save them in a file.

Basically, this application is like a Help uitlity for Java API classes. You will provide an interface for users to enter a class name. When they confirm their selection, a list of methods under the specified class will appear from which users can select any number of methods from the list. After making their selection from the list, users must confirm their choices. At this stage, details of the methods will be displayed for quick reference. Optionally, this information can also be saved to a file. For this chapter project it is enough to establish the basic framework and provide details of only one class. Populating the application repository with details of all API classes is up to you; however, if a user specifies a class whose details are not in the code, the application should inform the user about the omission.

In summary, components required for this application are as follows.

1. A text field for the user to specify the class name.
2. A button to confirm the class name specified.
3. A scrolling list to display the methods under the specified class enabling the user to make multiple selection of methods she or he desires.

4. A button to confirm the list of selections.
5. A text area where the method details will be displayed.
6. A button for the user to click when to save the displayed details into a file.

## Building the Project

**1.** First create a class apiReference which subclasses a Frame. It can contain a text field for the user to enter the class name, a list to display the method names under the class, a text area to display the method details obtained for the specified <class, methods> pair. The class also has a member to keep track of number of method names selected. Enter the following code in a file named apiReference.java.

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

public class apiReference extends Frame {

  TextField class_name;
  List meth_list;
  TextArea meth_desc;
  int selected;
  /* You will enter more code here */
}
```

**2.** Having created a basic framework for the class, you can now write its constructor to create an instance of the class apiReference. Decide on the classes you plan to use and the layout of the windowing components. Let the main Frame be in a BorderLayout. Group the components logically for better managment of the application frame. Initialize the class members declared in Step 1. Initialize the class_name member to an instance of TextField with 15 rows and the default class name Dialog. Create a panel for this text field and a button “ClassOK” to confirm the class name specified. Enter the following constructor code inside the class apiReference.

```
public apiReference() {
  setLayout(new BorderLayout());
    // now the layout of the class is BorderLayout
  selected = 0;
    // number of methods selected till now is zero.
  class_name = new TextField(“Dialog”, 15);

  Panel class_p = new Panel();
  class_p.setLayout(new FlowLayout());
  class_p.add(class_name);
         // adds the text field to the panel
  class_p.add(new Button(“ClassOK”));
         // adds a button to the panel to confirm class name
  add(“South”, class_p);
         //adds the panel, class_p, to the South of the main
Frame.
}
```

**3.** The next step is to create a list to display the method names. Let only five rows be visible in the List. Add a button and the multiple-selection list to a panel and

position the panel at the center of the application's main frame. Enter the following code inside the constructor defined in Step 2.

```
// inside the apiReference() constructor

   meth_list = new List(5, true);
        / 5 rows and allow multiple selectio
    Panel method_p = new Panel();
    method_p.setLayout(new BorderLayout());

            // add the list to the North of the panel
    method_p.add("North", meth_list);
    method_p.add("South", new Button("MethodOK"));
            // adds a button to the South of the panel
    add("Center", method_p);
            // add the panel to the center of the frame
```

**4.** Now you can create the text area for displaying the method details with a button to be pressed if the user wants to save the information in a file. Also, provide a menu "Quit," which has a menu item "Exit". When a user selects this, the application should end and the window should be closed. Enter the following code inside the apiReference constructor.

```
// create a text area for method description to appear

    meth_desc = new TextArea(10,15);
    Panel info_p = new Panel();
    info_p.setLayout(new BorderLayout());
    info_p.add("North", meth_desc);
    info_p.add("South", new Button("SaveInfo"));
          // a button for saving to a file
    add("East", info_p);       // add the panel to the east

           // a menu for the user to end the application and
           // dispose the window
    Menu quit = new Menu("Quit");
    quit.add("Exit");
    MenuBar mbar = new MenuBar();
    mbar.add(quit);
    setMenuBar(mbar);
    pack();
    show();
```

**5.** This application will be a stand-alone application, so the following main() method should be defined inside the apiReference class. Include the method definition in the class apiReference. After this step you can compile and run the application using the Java interpreter. This is done by typing "java apiReference" at the prompt. The interface will appear as shown in Figure 5-16, but the buttons won't yet be operative.

**Figure 5-16** The interface as it appears after Step 5

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

```
        apiReference inf = new apiReference();
        api.setTitle("Method Details App");
        api.pack();
        api.show();
  }
```

**6.** Now you need to write appropriate event handlers to handle user selections, deselections, and button presses. When the user selects or deselects an item from the method list, the value of variable "selected" should be changed to reflect the choice. When a button is pressed, an event ACTION_EVENT is generated. If it is the ClassOK button, then the text from the text field should be read. The methods in the selected class should be displayed in the list. When the user clicks on the MethodOK button, the details about that method are appended to the text area, thus displaying the method information. Whereas, if the user clicks on the SaveInfo button, a FileDialog should pop up enabling the user to specify a file name in which to save the information. The application should be stopped if a menu item is selected (there is only one "Exit" menu item). The event handling routine handleEvent takes care of all the above mentioned events. Include the following code in the apiReference class.

```
String classReq;      // string to get the class requested

 public boolean handleEvent(Event evt) {

    if (evt.id == Event.LIST_SELECT) {
       // if an item is selected from the list of methods
       selected++;
       return true;
    }

    if (evt.id == Event.LIST_DESELECT) {
       // if an item is deselected from the list of methods
       selected--;
       return true;
    }
 if (evt.id == Event.ACTION_EVENT) {
  // ACTION_EVENT is generated

     if (evt.target instanceof Button) {
          Button click = (Button)evt.target;

          if (click.getLabel().equals("ClassOK")) {
             // classOK buttton is pressed

             classReq = class_name.getText();
             displayMethodList(classReq);  // method to list the
                                              methods
                                           // in List
             return true;
          }
                if (click.getLabel().equals("MethodOK")) {
                  // MethodOK button is pressed
                String[] methods = meth_list.getSelectedItems();
                // for each of the selected methods, display
                   the details
                   for (int i=0; i < selected; i++) {
```

```
                    if (i==0)
                        meth_desc.setText(Method_Details(classReq,
                        methods[i]));
                        else {
                        meth_desc.appendText(new String("\n"));

meth_desc.appendText(Method_Details(classReq,
                        methods[i]))
                  }
          }
            return true;
          }

               if (click.getLabel().equals("SaveInfo")) {
                 // if the SaveInfo button is pressed to save file
                   FileDialog fd = new FileDialog(this, "File
Window",
                   FileDialog.SAVE);

                  fd.show();
                     String file_name = fd.getDirectory().trim() +
                     fd.getFile().trim();
                     System.out.println(" selected file is "
                     + file_name);
                    // after getting the filname from FileDialog,
save
                       the
                    // information on to a file using File I/O

                     File file = new File(file_name);
                     try{
                     DataOutputStream f_out = new
DataOutputStream(new
                                        BufferedOutputStream ( new

FileOutputStream(file_name)));
                      System.out.println(" text is " +
                      meth_desc.getText());
                      f_out.writeUTF(meth_desc.getText().trim());
                      f_out.flush();
                      f_out.close();
                    } catch (IOException io){}

                         return true;
                }
      }

if (evt.target instanceof MenuItem) {
      // Exit the application if a menuitem is selected
         MenuItem click = (MenuItem)evt.target;
             if (click.getLabel().equals("Exit"))
             System.exit(0);
  }
}

  return false;
}
```

**7.** Now implement the displayMethodList method. Given a class name, Dialog, all the methods in the class are displayed. If any other class is specified, a dialog window saying that the class details are not yet available is displayed. Enter the following method implementation in the apiReference class.

```
private void displayMethodList(String req_class) {

    if (req_class.equals("Dialog")) {
       meth_list.addItem("addNotify");
       meth_list.addItem("getTitle");
       meth_list.addItem("isModal");
       meth_list.addItem("isResizable");
       meth_list.addItem("paramString");
       meth_list.addItem("setResizable");
       meth_list.addItem("setTitle");
    }
    else {
       meth_list.clear();
       methodDialog dl = new methodDialog(this, false);
       dl.pack();
       dl.resize(250,100);
       dl.show();
   }
}
```

**8.** Next comes the code that implements Method_Details, as called in the MethodOK branch of button event handling. To save space, the details of each method are not provided. All you have to do is to supply the detail of each method as a String and return the string as follows:

```
public String Method_Details(String class_n, String method) {
   // if the class is not yet defined, pop-up a dialog window
     if (!class_n.equals("Dialog"))

       return new String("Sorry! Details of " + class_n
                + "not available now ;-( "
     else {
        // space savers ; you fill the details
         String details = new String(" ");
        String ret = new String(" Details of " + method +
                                are \n" + details );
      return ret;
    }
}
```

**9.** The dialog that pops up is defined as a methodDialog class with an OK button to close the dialog. Enter the following code which defines the class in the file apiReference.java. After entering this code, compile and run the application. Figure 5-17 shows the application in action. In Figure 5-17, the class chosen is Dialog and the two methods selected are addNotify and isModal. The details of the methods appear in the text area in the main frame. You can see a file dialog window that appears in front of the main frame, which shows two files in the directory named "doc" and the name of the file to which the details are to be saved is specified as methods.txt.

**Figure 5-17** The API Reference Interface Application in action

```
class methodDialog extends Dialog {
    public methodDialog(Frame parent, boolean modal) {
     super(parent, modal);
     setBackground(Color.gray);
     setTitle("Method Dialog");
     Panel ok = new Panel();
     ok.add(new Button("OK"));
     add("South", ok);
    }

    public boolean action(Event evt, Object arg) {

         if ("OK".equals(arg)) {
               dispose();
               return true;
         }
         return false;
    }
    public void paint(Graphics g) {
        g.setColor(Color.white);
        g.drawString("Method details NOT available ..sorry!", 20,
20);
    }
}
```

## How It Works

The API Reference Interface Application illustrates the use of TextArea, TextField, Dialog, FileDialog, List, and other windowing components. In this application, choices are confirmed using buttons in the interface. This application will form a very useful on-line reference for Java APIs.

When the application is started, you get a window with a TextField at the bottom where you can enter a class name. After you confirm your choice of class name using the ClassOK button, all the methods that form a part of that class are listed in the list area. The list area is at the left side of the interface. This list is a Java List object in multiple selection mode. You can get the details of as many methods of the class as you want, by selecting the appropriate method name from the list that is displayed. Once you confirm the list of methods you have selected by clicking on the MethodOK button, the information about what the method does will appear in the text area on the right side of the interface. You will note that all it gives right now is a string which says "Details of methodname are ". If you look up the implementation of the application, in Step 8, you have implemented a class named Method_Details. Presently, there is implementation for

only the Dialog class and that is very minimal. You can easily extend this project by entering details of more classes and details of each method of the class. This way you can have an on-line API reference for the Java APIs.

After obtaining the method details, you have an option of saving the details to a file for future reference. This is done by clicking on the Save button. This will bring up a file selection dialog box in which you can navigate around directories and specify a file name. On clicking the Save button in the FileDialog window, you effectively save the details to the file you have selected. During the process of using this application, if the system encounters a request which is not implemented, a Dialog window pops up to notify you of the problem.

This application makes effective use of the classes covered in this chapter. By proper extension of this application, you will not only gain a hands-on experience of this application's implementation, but also will know how to go about using the classes in a real-world application. You can exit from the application by pulling down the Quit menu that is available in the menu bar.

# *Part III*
# *Selection And Image Processing Tools*

## Chapter 6
## Choice, Menus, And Checkboxes

The objective of any user-interface is to make it easy for users to provide the input values that an application needs. Your goal is to provide the best possible combination of available functionality to make the interface user-friendly. Typing in input values for each and every parameter in an application is the last thing any user wants to do. Choice, menus, and checkbox related classes in the Java Development Kit (JDK) provide a comprehensive set of tools for building windowing applications which allow users to select inputs and actions easily as the program is running.

Devices, ranging from automated greeting card machines to automated bank tellers, offer users a sequence of selection options. When you call a company's customer service, you invariably listen to a series of messages offering selections like "If you are a Java programmer, press 1 now. If you are a C++ programmer, press 2 now. If you are a COBOL programmer please stay on the line and a representative will be with you in a few hours." Negotiating the selections can be maddening at times. Imagine someone whose car has broken down, calling a towing company, making a series of selections, and finally hearing "If you are in a deep pit press 1. If you are in a shallow pit press 2.". Obviously a thoughtfully designed selection interface that's responsive to its user's needs is important to a product's success.

Depending on the characteristics of an application, you can use menus, menu bars, menu items, choice buttons, checkboxes, and checkbox groups to develop such an interface. Java provides the classes Choice, MenuComponent, MenuBar, Menu, MenuItem, Checkbox, CheckboxGroup, and CheckboxMenuItem to implement these selection mechanisms in GUIs. Figure 6-1 shows an example window containing components that use these classes and their member functions.

**Figure 6-1** A typical window containing choice, menu, and checkbox buttons

In this chapter, we'll focus on these selection classes and their methods. As usual, we will provide class method summaries followed by detailed descriptions of each class, interface, and method. In the project for this chapter, you'll create the selection interface shown in Figure 6-1 with a little additional twist. You will add functionality to the interface, to display a specified string (or a default string) with a combination of font, color, and size. Now let's take a quick look at each of the three types of classes you'll be using.

## The Choice Class

The Choice class represents a set of pop-up choices in a user-interface. It subclasses the Component class in the Abstract Window Toolkit in Java. The functionalities of adding options, selecting an item among the choices, and finding the selected choice in a choice button are provided as member functions of this class. A choice button is a component that offers a list of options with the selected item displayed as the title of the button. This is helpful when there is a relatively large number of options, and one of them must be selected to provide input to the application. In Figure 6-1, a choice button is used for selecting the font size. The current selection of 12 is displayed as the title of the choice button. Only the selected item is displayed as the title of the choice button. The other items are hidden and become visible only when the user clicks on the button. Figure 6-2 illustrates another use of Choice. Three selections for the size of a pizza—small, medium, and large—are provided as a choice button. When the user selects Large by clicking on the button and dragging down to the Large option, it is displayed as the title of the button.


**Figure 6-2** Use of a choice button to specify the size of a pizza

## Menu-Related Classes

A menu enables you to make choices from a small, fixed set of options. The List class, which is used to deal with large numbers of choices, is described in Chapter 5. Small

pop-up menus are encapsulated into the Menu class in Java. It pops up a list of menu items when the user clicks on the menu title. In Figure 6-1, Color, Quit, and Help are Menu objects contained in a MenuBar. These menus contain instances of the MenuItem class, which form the list of options in the pull-down menu. Menu and MenuItem are classes that subclass MenuComponent, so they can be contained in a menu container. Menu containers are encapsulated in the MenuContainer interface. The classes, Menu and MenuBar, implement this interface and, hence, they can contain other menu components. The MenuComponent class in Java represents the super-class of all menu related classes in Java. Items are added to a menu either using their label or as instances of the MenuItem class. A menu can also contain submenus of type Menu. Menu items can be individually enabled or disabled from user selection. A disabled menu item is grayed and cannot be selected. Figure 6-3 shows a menu titled Appetizer, which contains three elements: GarlicBread, BreadSticks, and CheeseSticks. The item CheeseSticks is disabled.


**Figure 6-3** An Appetizer menu containing menu items

A menu item can also be a checkbox. Such an item should be an instance of the CheckboxMenuItem class. A CheckboxMenuItem can be selected, and will remain selected in the menu, until the user clicks on it again. A menu can contain many checkbox items, as shown in Figure 6-4. This menu titled Beverage contains the items Java, Tea, Juice, and Soda. A user can select none of the beverages or any number of them. A separator can be used to partition the items in a menu logically. In Figure 6-4, hot drinks are separated from cold drinks.


**Figure 6-4** A Beverage menu contains checkboxes as menu items

Just as many menu items can be contained in a menu, many menus can be grouped together and attached to a frame. The MenuBar class is the container that encapsulates this behavior in Java. A menu bar appears at the top of the window and is attached to a Frame object. In Figure 6-5, the menus Beverage, Appetizer, and Info are grouped in a menu bar. As is customary, the Help menu, or as in this case, the Info menu, occupies the rightmost position.


**Figure 6-5** A menu bar containing three menus

## Checkbox Classes

When you want to provide an option where the variable can take only one of two possible values, you can use a Checkbox. The button labeled Bold in Figure 6-1 is a Checkbox object which, when selected, will display the text in **bold** style. Two more checkboxes, labeled Fun and Simple, are grouped into a CheckboxGroup so that only one of them can be enabled at any time. A checkbox can have one of the two states, on or off (yes or no). When the state is on (yes), then the checkbox is selected. When it is not selected, it is in off (no) state. You can use a checkbox when a variable or input has a binary value. The Checkbox class in Java represents the checkbox in a windowing context.

Unlike menu or choice items, a checkbox need not have a label; however, depending on the application, you might want to set the label of a checkbox. Figure 6-6 shows a row of checkboxes. The user can select none or many of the options provided as checkboxes, namely: Pepperoni, Sausage, Onions, and Peppers. In the figure, Onions and Peppers have been selected by the user.

**Figure 6-6** Multiple checkboxes provided for selection

A set of checkboxes can be specified as a mutually exclusive group. The CheckboxGroup class can contain many checkboxes, but only one of them can be “on” at any time. Selection of one checkbox automatically switches the state of the others. Figure 6-7 shows a checkbox group that contains two checkboxes, Dine-In and Carry-Out, only one of which can be true at any instant. The Dine-In option is selected in the figure.


**Figure 6-7** Two mutually exclusive checkboxes in a checkbox group

## Choice, Menu, and Checkbox Summary

Table 6-1 summarizes the classes and interfaces necessary for developing user interfaces in Java using choice buttons, menus, and checkboxes.

**Table 6-1** Class and interface description of choice, menu, and checkbox

| Class/Interface name | Description |
|---|---|
| Choice | Represents a pop-up menu of choices in a user interface. |
| MenuComponent | Represents the super-class of all menu related components. |

| | |
|---|---|
| MenuBar | Encapsulates a menu bar bound to an application's frame. |
| Menu | Forms a component of a menu bar. |
| MenuItem | Represents a choice in a menu as a String item. |
| MenuContainer | A Java interface; a class implementing this interface forms the super-class of all menu-related container classes. |
| Checkbox | Encapsulates a user-interface element with a boolean state. |
| CheckboxGroup | Creates a group of mutually exclusive Checkbox items such that only one item can be "on" at a time. |
| CheckboxMenuItem | Produces a checkbox to represent a choice in a menu. |

## Choice

**Purpose**

Represents a pop-up menu of choices in a user-interface.

**Syntax**

public class Choice extends Component

**Description**

Choice is used to implement a pop-up menu of options from which the user can select only one. The selected item will form the header of the choice. When a user clicks on the header, a menu of options appears. Users can select one of the options by dragging down with the mouse till the desired selection is highlighted. The newly selected option then becomes visible and other available options are hidden. Figure 6-8 illustrates the inheritance relationship of class Choice.

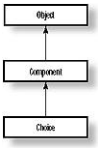


**Figure 6-8** Class diagram of the Choice class

**PackageName**

*java.awt*

**Imports**

*import java.awt.Choice;*

**Constructors**

public Choice()

**Parameters**

None.

**Example**

The class implemented in the following example, Listing 6-1, uses the class Choice and all the functions of this class. Enter the following code in a file named choice.java and compile it. Run it using the Java interpreter. Figure 6-9 shows the resultant window.


**Figure 6-9** Java application demonstrating use of Choice

**Listing 6-1** choice.java: Usage of Choice related methods

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

class choice extends Panel {

        Choice myc; // Choice component to appear in the window
public choice() {

        myc = new Choice();   // construct a Choice object
        myc.addItem("First");   // use addItem method to add
        items to the Choice
        myc.addItem("Second");
        myc.addItem("Third");
        myc.addItem("Fourth");

        add("Center", myc);   // place it at the center of the
        Panel

        int count = myc.countItems();
        System.out.println(" Number of Items in the Choice =
        " + count);
        System.out.println(" Item at location 2 is " +
        myc.getItem(2));
        System.out.println(" Selected item is at location " +
        myc.getSelectedIndex());
        System.out.println(" Selected item is  " +
        myc.getSelectedItem());

        System.out.println(" Going to select 3rd item");
        myc.select(3);
        System.out.println(" Going to select item labeled Fourth ");
        myc.select("Fourth");

}

public static void main(String args[]) throws IOException {

        choice c = new choice();
```

```
        Frame f = new Frame("choice demo");
        f.add("Center", c);
        f.pack();
        f.resize(200,300);
        f.show();
}
}
```

## addItem(*String*)

**ClassName**

Choice

**Purpose**

To add an item to the target Choice object.

**Syntax**

public synchronized void addItem(String item)

**Parameters**

***item***

The String representing the item to be added to the Choice.

**Description**

This method adds an item, with the label passed as parameter, to the list of options in this Choice. The item is added to the end of the list. This method is synchronized, so when multiple Threads are trying to add items to the target Choice object, only one Thread can add an item at a given instant.

**Imports**

*import java.awt.Choice;*

**Returns**

None.

**See Also**

The Choice class

**Example**

Refer to Listing 6-1, defining the class choice in Choice class description.

## addNotify()

**ClassName**

Choice

**Purpose**

This method creates a peer for the Choice. It helps to change the look of the Choice without changing its behavior.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

This method helps change the look of the Choice without changing its behavior. It overrides the addNotify() method of class Component. It creates a peer for this Choice.

**Imports**

*import java.awt.Choice;*

**Returns**

None.

**See Also**

The java.awt.peer.Choice class; Component class

**Example**

Refer to the example code, Listing 6-1, defining the class choice in Choice class description.

## countItems()

**ClassName**

Choice

**Purpose**

To determine the number of items in the Choice.

**Syntax**

public int countItems()

**Parameters**

None.

**Description**

The method counts the number of items in the Choice. The count is automatically increased when a new item is added. It is also the maximum index in this Choice.

**Imports**

*import java.awt.Choice;*

**Returns**

Return type is int. It is the number of items in the Choice at a given instance.

**See Also**

The getItem() method in class Choice

**Example**

Refer to Listing 6-1, defining the class choice in Choice class description. The variable count represents the number of items in the choice.

## getItem(*int*)

**ClassName**

Choice

**Purpose**

To obtain the item with the specified index.

**Syntax**

public String getItem(int index)

**Parameters**

***index***

The index of the item in the Choice to be retrieved.

**Description**

This method returns the String that represents the item, at the specified index, in this Choice. The index starts at 0; hence the index of the first item is 0, second item is 1 and so on. If the specified index is greater than or equal to the number of items, an ArrayOutOfBounds exception is thrown at runtime.

**Imports**

*import java.awt.Choice;*

**Returns**

The item at the specified index in its String form.

**See Also**

The Choice class; method addItem() in class Choice

**Example**

Refer to Listing 6-1, defining the class choice in Choice class description. The item Third, located at index two, is obtained using this method.

## getSelectedIndex()

**ClassName**

Choice

**Purpose**

To obtain the index of the selected item in the Choice.

**Syntax**

public int getSelectedIndex()

**Parameters**

None.

**Description**

When an item in the Choice is selected, it is displayed as the title of the menu. Other items are not visible until the user pulls down this Choice menu to view other items. Effectively, this method returns the index of the item that appears as the title of this Choice. The index starts from 0; so if the selected item is in tenth position in the menu, this method returns the value 9.

**Imports**

*import java.awt.Choice;*

**Returns**

The index which is of type int.

**See Also**

The Choice class; method getSelectedItem() in class Choice

**Example**

Refer to Listing 6-1, defining the class choice in Choice class description. By default getSelectedIndex selects the first item in the menu which is at index 0; so invoking this method in that example would return 0.

## getSelectedItem()

**ClassName**

Choice

**Purpose**

To obtain the selected item in its String representation.

**Syntax**

public String getSelectedItem()

**Parameters**

None.

**Description**

This method returns the string representation of the item that is selected. Note that you can achieve the same result by first using the getSelectedIndex() method to obtain the index of the selected item, and then passing this index to the method getItem() and obtaining the String representation.

**Imports**

*import java.awt.Choice;*

**Returns**

The return type is String and it is the string representation of the selected item.

**See Also**

The Choice class; the getSelectedIndex()method

**Example**

Refer to Listing 6-1, defining the class choice in Choice class description. The first item in a choice is selected by default. Usage of this method in the example returns the item labeled First.

## paramString()

**ClassName**

Choice

**Purpose**

To obtain the parameter String of this Choice.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method returns the String representation of this Choice containing its label, the selected item, and the class information. This method is protected, and hence can be used only by the classes within the *java.awt* package.

**Imports**

*import java.awt.Choice;*

**Returns**

The parameter string of type String.

**See Also**

The Choice and Component classes

**Example**

The following code uses paramString by subclassing the Choice.

```
package java.awt;
import java.awt.Choice;
```

```
class myChoice extends Choice {

       public myChoice() {
        super();
       }

        public String getmyStringForm() {
                        return super.paramString();
        }

        public static void main(String[] args) {
              myChoice ch = new myChoice();
              System.out.println(ch.getmyStringForm());
        }
}
```

## select(*int*)

**ClassName**

Choice

**Purpose**

To select the item whose position is specified.

**Syntax**

public synchronized void select(int index)

**Parameters**

***index***

The index of the selected item in the Choice.

**Description**

This method selects the item specified by the position as the parameter. The internal representation of the Choice will recognize the item to be selected. This is a synchronized method, as the position in the Choice table is shared data and use of this data by different methods should be consistent. If the specified position is out of bounds, this method throws an IllegalArgumentException. This need not be caught using the try and catch construct in Java, as it forms a runtime exception and hence, is not declared to be thrown.

**Imports**

*import java.awt.Choice;*

**Returns**

None.

**See Also**

The Choice class; methods getSelectedItem(), and getSelectedIndex()

**Example**

Refer to Listing 6-1, defining the class choice in Choice class description. The item whose index is 3 is selected in the example.

## select(String)

**ClassName**

Choice

**Purpose**

To add an item to the target Choice object.

**Syntax**

public void select(String item)

**Parameters**

***item***

The String representing the item to be selected.

**Description**

This method selects the item whose String representation matches the parameter passed. Note that this method need not be synchronized, as you are not accessing any shared data (such as the position in the Choice).

**Imports**

*import java.awt.Choice;*

**Returns**

None.

**See Also**

The Choice class; methods getSelectedItem() and getSelectedIndex() in class Choice

**Example**

Refer to Listing 6-1, defining the class choice in Choice class description. The item labeled Fourth is selected in the example.

## MenuComponent

**Purpose**

Represents the super class of all menu-related components.

**Syntax**

public class MenuComponent extends Object

**Description**

The common characteristics of all menu related components are encapsulated into this class. The common functionalities, namely setting and getting the font, obtaining a handle to the parent container, posting an event to the component, getting the String representation of the menu components, and handling the peers, are grouped into this super class. You can use a reference to this class when you need to remove a menu component from a menu or menu bar, and when you do not know in advance what component you are removing. The classes Menu, MenuBar, and MenuItem are ultimately subclasses of this class. Figure 6-10 illustrates the inheritance relationship of class MenuComponent.

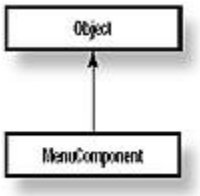


**Figure 6-10** Class diagram of MenuComponent class

**PackageName**

*java.awt*

**Imports**

*import java.awt.MenuComponent;*

**Constructors**

public MenuComponent()

**Parameters**

None.

**Example**

The following example in Listing 6-2 includes instances of MenuComponent, Menu, MenuBar, and MenuItem. All the methods under these classes are also used in this example. You can enter this source in a file named myMenu.java and compile it. Figure 6-11 shows the resultant output on execution (except for the output you see on the terminal giving some information). All the methods in the above mentioned classes refer to this example for their usage. This example shows how these methods can be used effectively. The Quit menu is tearable. On clicking the Exit item, an event is posted to destroy the window. Appropriate action is taken by the event handler. The Rename item under the File menu should be disabled from user selection when this code is executed.


**Figure 6-11** Resultant window demonstrating the usage of menu-related classes and methods

**Listing 6-2** myMenu.java: Usage of all menu-related methods

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

class myMenu extends Frame {
        // you are going to add a Menu bar and so this class
        should subclass Frame.

        MenuBar mbar; // a member of type MenuBar
     public myMenu() {

// Illustration of methods of class Menu
```

```
        Menu file_m = new Menu("File");
        MenuItem load = new MenuItem("Load");
        load.enable();

        file_m.add(load);
        MenuItem save = new MenuItem("Save");
        file_m.add(save);

        MenuItem ren = new MenuItem("Hi");
        ren.setLabel("Rename");
        System.out.println(" the menu ren is labeled " +
        ren.getLabel());
        ren.disable();
        if (ren.isEnabled())
              System.out.println(" The disable function did not
work!");
        file_m.add(ren);

        file_m.addSeparator();
        file_m.add("Close");

        System.out.println(" Parent of load menu item is " +
        load.getParent().toString());

        Menu quit = new Menu("Quit", true);
        quit.add("Exit");
        quit.add("Hide");  // will remove this item soon from the
        menu

        System.out.print(" Quit menu is ");
        if (!quit.isTearOff())
              System.out.println("NOT");
        System.out.println(" a tear-off menu ");
        quit.remove(1);

        Menu tmp = new Menu("Tmp"); //temporary menu; will be
        removed

        Menu info = new Menu("Info");
        info.add("About");

// MenuBar methods illustration
        mbar = new MenuBar();
        mbar.add(file_m);
        mbar.add(quit);
        mbar.add(info);
        mbar.add(tmp);
        mbar.setHelpMenu(info);
        setMenuBar(mbar);
        System.out.println(" Number of menus in menu bar is " +
        mbar.countMenus());
        System.out.println(" Help menu is " +
        mbar.getHelpMenu().toString());

        System.out.println(" Menu at index 1 is " +
        mbar.getMenu(1).toString());
```

```
         mbar.remove(tmp);

        // run initially without un-commenting the "remove" method;
        // next time, un-comment the line, compile and run.
        // after that change 3 to 2 and observe the change
        // mbar.remove(3);

// MenuComponent methods illustration
        System.out.println(" Parent of Quit menu is " +
        quit.getParent().toString());

        System.out.println(" paramString of File menu is " +
        file_m.paramString());

        info.setFont(new Font("myFont", Font.BOLD, 16));

        System.out.println(" Font of Info menu is " +
        info.getFont());
        }

public boolean action(Event evt, Object arg) {

        if (evt.target instanceof MenuItem) {
          MenuItem m_it = (MenuItem)evt.target;
             if ("Exit".equals(arg)) {
             // you can use System.exit(0) here but try throwing
             an event
             // and handling it using handleEvent() method

             Event exit_event = new
             Event(m_it,Event.WINDOW_DESTROY,mbar);
             m_it.postEvent(exit_event);
             System.out.println(" Event posted");
             }
        }
        return true;

}

// Method to handle events. This is used here to illustrate the
usage of postEvent method.
// the posted event is handled in this method and appropriate
action is taken.

public boolean handleEvent(Event e)
    {
        // Window Destroy event
        if (e.id == Event.WINDOW_DESTROY)
        {
           // exit the program
           System.exit(0);
           return true;
        }
        // it's good form to let the super class look at any
        // unhandled events
        return super.handleEvent(e);
```

```
    }  // end handleEvent()

 public static void main(String args[]) {
        myMenu m = new myMenu();
        m.setTitle("menu demo");
        m.pack();
        m.resize(200,200);
        m.show();
 }
}
```

## getFont()

**ClassName**
MenuComponent
**Purpose**
To determine the font used for the target MenuItem object.
**Syntax**
public Font getFont()
**Parameters**
None.
**Description**
The current font, to which the menu item is set, is returned by this method when it is invoked on the menu item. This font corresponds to the style and size of the String label that represents the menu item. Use setFont method to set or change the font. It returns null, if it has not been set to any Font earlier.
**Imports**
*import java.awt.MenuComponent;*
**Returns**
The font of the menu item; the return type is Font.
**See Also**
The MenuComponent class; the setFont() method of the MenuComponent class
**Example**
Refer to Listing 6-2 defining the class myMenu in MenuComponent class description. In the example, after setting the font for the Info menu to **bold** of size 16, this method is used to check it.

## getParent()

**ClassName**
MenuComponent
**Purpose**
To disclose the parent container object of the target MenuComponent object.
**Syntax**
public MenuContainer getParent()
**Parameters**

None.

**Description**

A menu item or component is contained in a menu container. Any class that implements the interface MenuContainer forms an instance of MenuContainer and hence can be treated as an object. Thus instances of Menu and MenuBar can be considered as objects of type MenuContainer. The parent of a component is the object (Menu or MenuBar, for example) to which this component is added. This is helpful when you handle an event and the target you obtain is MenuItem; you can determine which Menu it belongs to by using this method.

**Imports**

*import java.awt.MenuComponent;*

**Returns**

The parent in which this component is contained.

**See Also**

The MenuContainer interface

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. The parent of the Load menu item is obtained using this method and this method returns the File menu.

## getPeer()

**ClassName**

MenuComponent

**Purpose**

To obtain the peer of this menu component.

**Syntax**

public MenuComponentPeer getPeer()

**Parameters**

None.

**Description**

A peer is created for a component on creation or when addNotify() is called. This method gets the MenuComponentPeer that is created for this component. The peer allows you to modify the appearance of the menu component without changing its functionality. Use this method only if you are writing your own AWT implementation.

**Imports**

*import java.awt.MenuComponent;*

**Returns**

The return type of this method is MenuComponentPeer.

**See Also**

The MenuComponentPeer class; the addNotify methods of subclasses of MenuComponent

**Example**

Refer to Chapter 9 on AWT Peer Interfaces.

## paramString()

**ClassName**

MenuComponent

**Purpose**

The String parameter of this MenuComponent is returned.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method is used to obtain the String parameter of the MenuComponent, the parameters of a Component object. This is a protected method and, hence, this method can only be called within the *java.awt* package.

**Imports**

*package java.awt;*
*import java.awt.MenuComponent;*

**Returns**

The String parameter of this MenuComponent.

**Example**

The following code uses paramString by subclassing the MenuComponent.

```
package java.awt;
import java.awt.MenuComponent;

class myMenuC extends MenuComponent {

         public myMenuC() {
         super();
        }

        public String getmyStringForm() {
                        return super.paramString();
         }

         public static void main(String[] args) {
               myMenuC mC = new myMenuC();
               System.out.println(mC.getmyStringForm());
         }
}
```

## postEvent(Event)

**ClassName**

MenuComponent

**Purpose**

To post the specified event to the menu.

**Syntax**

public boolean postEvent(Event evt)

**Parameters**

***evt***

The event which you want to take place on this MenuComponent.

**Description**

The specified event is posted to the menu when this method is called. The Event encapsulates the target object and the type of event that is thrown. For example, when a user selects Menu item (which is a MenuComponent), you can throw an event using this method. If appropriate action is specified in the handleEvent() method, then the purpose of throwing the event is satisfied.

**Imports**

*import java.awt.MenuComponent;*

**Returns**

The return type of this method is boolean; it returns true if the specified event is successfully posted and false if it is not successfully posted.

**See Also**

The Event class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. When the user selects the Exit menu item under the Quit menu, a window destroy event is posted. This is handled by the handleEvent method.

## removeNotify()

**ClassName**

MenuComponent

**Purpose**

To remove the peer of this menu component.

**Syntax**

public void removeNotify()

**Parameters**

None.

**Description**

A menu component peer is used to change the appearance of your menu component without changing its functionality. This method removes the component's peer.

**Imports**

*import java.awt.MenuComponent;*

**Returns**

None.

**See Also**

The MenuComponentPeer class; the addNotify methods of subclasses of MenuComponent

## setFont(Font)

**ClassName**

MenuComponent

**Purpose**

Sets the font of the target component object to the specified font.

**Syntax**

public void setFont(Font fnt)

**Parameters**

***fnt***

The font to which you want your component to be set.

**Description**

The current font of the menu item is set to the specified Font *fnt*. This font corresponds to the style and size of the String label that represents the menu item. Use getFont method to get the font of the component.

**Imports**

*import java.awt.MenuComponent;*

**Returns**

None.

**See Also**

The getFont()method of the MenuComponent class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. In the example, font for the Info menu is set using this method.

## toString()

**ClassName**

MenuComponent

**Purpose**

The String representation of the values of this MenuComponent s returned.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

The details related to this MenuComponent are returned in a string form. It is useful for debugging purposes.

**Imports**

*import java.awt.MenuComponent;*

**Returns**

The information on this component in string form.

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. In the example, this method is used to get the String form of the parent menu of load.

## MenuBar

**Purpose**

Encapsulates the concept of a menu bar bound to a Frame.

**Syntax**

public class MenuBar extends MenuComponent implements MenuContainer

**Description**

The MenuBar class encapsulates the platform's concept of a menu bar bound to a Frame. It is the list of visible menu components which usually appears at the top of the frame in any user interface. A pull-down menu will appear when one of the menu items in the list is clicked by the user. A MenuBar is associated with the Frame using the setMenuBar() method in the class Frame. Because MenuBar can only be attached to a Frame, you can subclass Frame and set the menu bar in this class. Another way is to invoke the setMenuBar on an appropriate Frame object. The MenuBar class contains the methods to add a menu to the menu bar, to remove a menu, to get a specified menu and to get a help menu. This class is a MenuComponent by itself, as it extends the MenuComponent class. Hence, though the methods to set Font and to get Parent are not listed under this class, these are inherited from the super class MenuComponent. Also, this class implements the MenuContainer interface and thus, acts as a container which can contain other menu components. Figure 6-12 illustrates the inheritance relationship of class MenuBar.

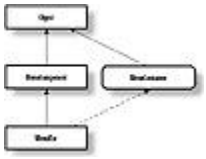

**Figure 6-12** Class diagram of the MenuBar class

**PackageName**

*java.awt*

**Imports**

*import java.awt.MenuBar;*

**Constructors**

public MenuBar()

**Parameters**

None.

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. Members of the class menu are of type MenuBar (variable *mbar*). This variable is instantiated inside the constructor of menu.

## add(*Menu*)

**ClassName**

MenuBar

**Purpose**

Add the specified Menu to this MenuBar.

**Syntax**

public synchronized Menu add(Menu m)

**Parameters**

***m***

The menu that has to be added to the MenuBar.

**Description**

After MenuBar has been constructed, it is populated with menus and related components using this method. For every addition of a menu, the index of the MenuBar is incremented. The menus are added from left to right when the MenuBar appears at the top of the user-interface. This method is synchronized, so only one Thread of control can add a menu at a given instant.

**Imports**

*import java.awt.MenuBar;*

**Returns**

The return type of this method is Menu. It returns a handle to the Menu, added to this MenuBar for referencing at a later time.

**See Also**

The Menu class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. In this example, using this add method, the menus file_m, quit, info, and tmp are added to the MenuBar object, *mbar*.

## addNotify()

**ClassName**

MenuBar

**Purpose**

This method creates a peer of this MenuBar.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

An instance of the MenuBarPeer is created as a peer for this MenuBar item. Using the peer you can change the appearance of the MenuBar without modifying its functionality.

**Imports**

*import java.awt.MenuBar;*

**Returns**

None.

**See Also**

The MenuBarPeer class

**Example**

Refer to Chapter 9 describing the peers and interface for more details.

## countMenus()

**ClassName**
MenuBar
**Purpose**
This method returns the number of menus in this MenuBar.
**Syntax**
public int countMenus()
**Parameters**
None.
**Description**
Whenever you use add method to add a menu to the MenuBar, the count is incremented. This method helps you to obtain the number of Menus in this MenuBar which is also the number of labels in the MenuBar.
**Imports**
*import java.awt.MenuBar;*
**Returns**
The number of menus in the MenuBar; the return type is int.
**See Also**
The Menu class
**Example**
Refer to Listing 6-2, defining the class menu in MenuComponent class description. In this example, this method is invoked on the member *mbar*.

## getHelpMenu()

**ClassName**
MenuBar
**Purpose**
This method returns the menu which is set as the HelpMenu in the MenuBar.
**Syntax**
public Menu getHelpMenu()
**Parameters**
None.
**Description**
The help menu in this MenuBar is accessed using this method. It returns the menu set by the setHelpMenu method. If you know the index of the Help menu you can also accomplish this using the getMenu method.
**Imports**
*import java.awt.MenuBar;*
**Returns**
An object of type Menu, which is the help menu in this MenuBar.
**See Also**
The setHelpMenu and getMenu methods of the MenuBar class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. In the example, the menu object *info* is set as Help menu. This method returns a handle to the info menu when invoked on the menu bar.

## getMenu(*int*)

**ClassName**

MenuBar

**Purpose**

This method returns the Menu at the specified index.

**Syntax**

public Menu getMenu(int index)

**Parameters**

***index***

The index of the menu to be returned.

**Description**

The Menus in the MenuBar can be referenced using their respective index. This method gets the Menu when you specify its index as the parameter. Index starts at 0. To obtain the third Menu in the menu bar, you should index it as 2.

**Imports**

*import java.awt.MenuBar;*

**Returns**

The Menu at the specified index.

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. In the example, the Menu at index 1 is obtained by using this method. The name of the menu is printed by obtaining the string form of the Menu.

## remove(int)

**ClassName**

MenuBar

**Purpose**

This method removes the Menu at the specified index in this MenuBar.

**Syntax**

public synchronized void remove(int index)

**Parameters**

***index***

The index of the Menu to be removed from the MenuBar.

**Description**

This method removes a Menu from the MenuBar at the specified index. Index starts from 0. For example, to remove the third menu from the menu bar, specify the index as 2. If the specified index is greater than or equal to the number of menus in the menu bar, an ArrayOutOfBoundsException is thrown at runtime. After removing the Menu, changes are made to the MenuBar structure to reflect

the removal. This includes the decrement in the count and updating other indices. To ensure this is performed in a consistent manner by the runtime, this method is declared a synchronized method.

**Imports**

*import java.awt.MenuBar;*

**Returns**

None.

**See Also**

The add(Menu) method of the MenuBar class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. The menu tmp is removed from the MenuBar, mbar, by specifying its index as 3. Un-comment the line mbar.remove(3) in the example code and observe the change.

## remove(MenuComponent)

**ClassName**

MenuBar

**Purpose**

This method removes the Menu specified by the MenuComponent parameter.

**Syntax**

public synchronized void remove(MenuComponent mc)

**Parameters**

***mc***

The reference to the MenuComponent object to be removed.

**Description**

This method removes a Menu specified by the reference in the parameter from the MenuBar. After the menu is removed, changes are made to the MenuBar structure to reflect the removal. This includes decrementing the count and updating other indices. To ensure these tasks are performed in a consistent manner by the runtime, this method is declared a synchronized method.

**Imports**

*import java.awt.MenuBar;*

**Returns**

None.

**See Also**

The method add(Menu) in class MenuBar

**Example**

Refer to Listing 6-2, defining the class menu in MenuComponent class description. In the example, the object tmp is removed from the menu bar using this method.

## removeNotify()

**ClassName**

MenuBar

**Purpose**

This method removes the peer of this MenuBar.

**Syntax**

public void removeNotify()

**Parameters**

None.

**Description**

Removes an instance of the MenuBarPeer, which was a peer for this MenuBar item. Using the peer you can change the appearance of the MenuBar without modifying its functionality.

**Imports**

*import java.awt.MenuBar;*

**Returns**

None.

**See Also**

The MenuBarPeer class; the addNotify()method of the MenuBar class

**Example**

Refer to the details in Chapter 9 describing the peers and interfaces.

## setHelpMenu(Menu)

**ClassName**

MenuBar

**Purpose**

This method sets the Help menu in the MenuBar.

**Syntax**

public synchronized void setHelpMenu(Menu mnu)

**Parameters**

***mnu***

The menu, of type Menu, which is to be added to this MenuBar.

**Description**

The specified menu is set as the help menu in this MenuBar, using this method. It places the menu at the top-right corner in a frame, i.e., at the right end of a Menubar. This is the place where the Help menu is made available in all interfaces. This method is synchronized; hence, only one Thread can be running this method at a given instant.

**Imports**

*import java.awt.MenuBar;*

**Returns**

None.

**See Also**

The getHelpMenu and getMenu methods of the MenuBar class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. The menu object *info* is set to be the help menu in the example. A call

to this method sets *info* to be the help menu and places the info menu (labeled "Info"), at the rightmost end of the menu bar.

## Menu

**Purpose**

A Menu serves as a component of a menu bar.

**Syntax**

public class Menu extends MenuItem implements MenuContainer

**Description**

A menu bar contains many menus. An instance of class Menu is created with an associated label. It can be identified by its label or by using the index of its location in a menu bar. You can specify any menu as a tear-off menu. This means that the menu will remain visible on the screen even after you release the mouse button. A frequently used menu can be declared tear-off so that the users need not pull the menu down from its title every time and then select. A menu which is not a tear-off menu disappears from the screen when the mouse button is released. This class includes methods to add menu items, to add separators, to count items, to get items, and to remove items. It is a subclass of MenuItem and, hence, an instance of this class can be a part of some menu container. This class also implements the MenuContainer interface; it can contain other menu components. Figure 6-13 illustrates the inheritance relationship of class Menu.

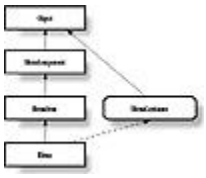

**Figure 6-13** Class diagram of the Menu class

**PackageName**

*java.awt*

**Imports**

*import java.awt.Menu;*

**Constructors**

public Menu(String label)

public Menu(String label, boolean tearOff)

**Parameters**

***label***

The string associated with this menu.

***tear-off***

The boolean value specifying whether or not you want your Menu to be a tear-off menu.

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. Four menus are created in the example, namely, file_m, quit, tmp, and info.

## add(MenuItem)

**ClassName**

Menu

**Purpose**

This method adds the specified menu item to the target menu object.

**Syntax**

public synchronized MenuItem add(MenuItem item)

**Parameters**

***item***

The item of type MenuItem to be added to this menu.

**Description**

A menu consists of many menu items. Those items can be added to the menu using this method. A handle to this menu item can be obtained by using the getItem method. This method is synchronized; so only one Thread can be running this method at a given instant to add an item to the Menu.

**Imports**

*import java.awt.Menu;*

**Returns**

A reference to the added MenuItem

**See Also**

The add(String), remove(int), and remove(MenuComponent) methods of the Menu class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. A menu item labeled "Load" is created as a MenuItem and then placed in the menu file_m.

## add(*String*)

**ClassName**

Menu

**Purpose**

This method adds an item with the specified String label to this Menu.

**Syntax**

public synchronized void add(String label)

**Parameters**

***label***

The label of the item to be added to this Menu.

**Description**

A Menu consists of many menu items with a unique label identifying each of them. Those menu items can be added to a menu using this method. A handle to any item can be obtained using the getItem method. Because this method is

synchronized, only one Thread can be running this method at a given instant to add an item with a specified label to the Menu.

**Imports**

*import java.awt.Menu;*

**Returns**

None.

**See Also**

The add(MenuItem), remove(int), and remove(MenuComponent) methods of the Menu class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. In the example, a menu item in the menu file is added using this method with the label “Save”.

## addNotify()

**ClassName**

Menu

**Purpose**

This method creates a peer for this Menu.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

You can change the appearance of a menu using the peer created by this method. This method overrides the addNotify() method of the super class, MenuItem.

**Imports**

*import java.awt.Menu;*

**Returns**

None.

**See Also**

The MenuPeer interface; the removeNotify()method of the Menu class

**Example**

Refer to Chapter 9 describing the peers and interface for more details.

## addSeparator()

**ClassName**

Menu

**Purpose**

This method adds a separator line or a hyphen at the specified position.

**Syntax**

public void addSeparator()

**Parameters**

None.

**Description**

A Menu consists of many menu items. To help the user scan the list of options in a menu easily you can add a separator between groups of logically related menu items. You can use this method to add a separator line or a hyphen at any position.

**Imports**

*import java.awt.Menu;*

**Returns**

None.

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. A separator is added before the "Close" menu item in the menu file_m in that example. Also see Figure 6-4, where beverages are listed with a separator between hot and cold.

## countItems()

**ClassName**

Menu

**Purpose**

To obtain the number of items in a menu.

**Syntax**

public int countItems()

**Parameters**

None.

**Description**

The method counts the number of items in the Menu. The count is automatically increased when a new item is added, and becomes the maximum index in this menu.

**Imports**

*import java.awt.Menu;*

**Returns**

The number of items in the Menu; return type is int.

**See Also**

The getItem()method in the Menu class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description.

## getItem(int)

**ClassName**

Menu

**Purpose**

This method returns the item with the specified index.

**Syntax**

public MenuItem getItem(int index)

**Parameters**

***index***

The index of the item in this Menu to be retrieved.

**Description**

The method returns the MenuItem object at the specified index in this Menu. Index value starts from 0; hence, the index of the third item is 2, the fourth item is 3 and so on.

**Imports**

*import java.awt.Menu;*

**Returns**

The item of type MenuItem at the specified index.

**See Also**

The Menu class; the addItem()method of the Menu class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description.

## isTearOff()

**ClassName**

Menu

**Purpose**

Determine whether the target Menu object is a tear-off menu or not.

**Syntax**

public boolean isTearOff()

**Parameters**

None.

**Description**

The method determines whether this menu is a tear-off menu. A tear-off Menu is one which stays visible on the screen even after the user releases the mouse button.

**Imports**

*import java.awt.Menu;*

**Returns**

Return type is boolean. True means this menu is a tear-off menu; false means it is not.

**See Also**

The Menu(String label, boolean tear-off) constructor

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. The quit menu is created as a tear-off menu in the example. Using this method, it is verified that the quit menu is indeed a tear-off menu.

## remove(int)

**ClassName**

Menu

**Purpose**

This method removes the item at the specified index position in this Menu.

**Syntax**

public synchronized void remove(int index)

**Parameters**

***index***

The index of the item to be removed from the Menu.

**Description**

This method removes the item from the Menu at the specified index. The index starts from 0. The first item has an index 0, the second item has an index 1 and so on. If the specified index is greater than or equal to the number of items in this Menu, an ArrayOutOfBoundsException is thrown at runtime. After the item is removed, the Menu structure is changed to reflect the removal. This includes decrementing the count and updating other indices. To perform this in a consistent manner by runtime, this method is declared a synchronized method.

**Imports**

*import java.awt.Menu;*

**Returns**

None.

**See Also**

The add(MenuItem) and add(String) methods of the Menu class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. After the menu quit is created, in the example, two items are added to it: Exit and Hide. The item Hide is removed using this method with the index being 1.

## remove(MenuComponent)

**ClassName**

Menu

**Purpose**

This method removes the item specified by the MenuComponent parameter.

**Syntax**

public synchronized void remove(MenuComponent mc)

**Parameters**

***mc***

The reference to the MenuComponent object to be removed.

**Description**

This method removes from the menu an item specified by the reference in the parameter. After the item is removed, changes are made to the Menu structure to reflect the removal. This includes decrementing the count and updating other indices. To perform this in a consistent manner by runtime, this method is declared a synchronized method.

**Imports**

*import java.awt.Menu;*

**Returns**

None.

**See Also**

The add(MenuItem) and add(String) methods of the Menu class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. A menu tmp is created temporarily and is deleted from the menu bar using this method.

## removeNotify()

**ClassName**

Menu

**Purpose**

This method removes the peer of target Menu object.

**Syntax**

public void removeNotify()

**Parameters**

None.

**Description**

Removes an instance of the MenuPeer which was a peer for this MenuBar item. Using the peer, you can change the appearance of the Menu without modifying its functionality.

**Imports**

*import java.awt.Menu;*

**Returns**

None.

**See Also**

The MenuPeer class; the addNotify()method of the Menu class

**Example**

Refer to Chapter 9 describing the peers and interface for more details.

## MenuItem

**Purpose**

Represents a choice as a string item in a menu

**Syntax**

public class MenuItem extends MenuComponent

**Description**

A menu contains a list of choices for the user. These choices are represented in their string form and are presented as objects of type MenuItem. MenuItem is a menu component by its inheritance relationship with MenuComponent class. Figure 6-14 illustrates the inheritance relationship of class MenuItem.

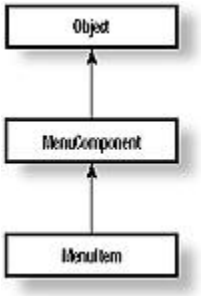


**Figure 6-14** Class diagram of the MenuItem class

**PackageName**
*java.awt*
**Imports**
*import java.awt.MenuItem;*
**Constructors**
public MenuItem()
**Parameters**
None.
**Example**
Refer to Listing 6-2, defining the class menu in MenuComponent class description.

## addNotify()

**ClassName**
MenuItem
**Purpose**
This method creates a peer for this MenuItem.
**Syntax**
public synchronized void addNotify()
**Parameters**
None.
**Description**
The appearance of a menu item can be changed using the peer created by this method.
**Imports**
*import java.awt.MenuItem;*
**Returns**
None.
**See Also**
The MenuItemPeer interface; the removeNotify()method of the MenuItem class
**Example**
Refer to Chapter 9 describing the peers and interface for more details.

## disable()

**ClassName**

MenuItem

**Purpose**

This method disables this MenuItem so that it cannot be selected from the menu.

**Syntax**

public void disable()

**Parameters**

None.

**Description**

A menu item is, by default, enabled on construction. Invoking this method on the menu item object will disable any action on this item. After this method is called, this menu item will be grayed and the user will not be able to select it. The enable() method must be invoked on this menu item to make it again selectable by the user.

**Imports**

*import java.awt.MenuItem;*

**Returns**

None.

**See Also**

The enable method of class MenuItem.

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. The Rename item in the File menu is disabled using this method.

## enable()

**ClassName**

MenuItem

**Purpose**

This method enables this MenuItem, making it selectable from the menu.

**Syntax**

public void enable()

**Parameters**

None.

**Description**

A menu item is, by default, enabled on construction. Invoking the disable() method on the menu item, disables any action on the item. You use this method to explicitly enable a previously disabled item. After this method is invoked, this MenuItem will be selectable by the user.

**Imports**

*import java.awt.MenuItem;*

**Returns**

None.

**See Also**

The enable(boolean) and disable()methods of the MenuItem class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. The Load menu item in the menu file_m in the example is explicitly enabled using this method.

## enable(boolean)

**ClassName**

MenuItem

**Purpose**

This method enables this MenuItem if the specified boolean condition is true.

**Syntax**

public void enable(boolean condn)

**Parameters**

***condn***

The condition which, when evaluated, has a boolean value; depending on the value the MenuItem is enabled or disabled.

**Description**

A menu item is by default enabled on construction. Invoking the disable() method on the menu item disables any action on this item. You can use this method to explicitly enable a previously disabled item depending on a specified boolean condition. After this method is invoked, this MenuItem will be selectable by the user if the boolean condition evaluates to true; if the condition evaluates to false, the MenuItem will remain disabled.

**Imports**

*import java.awt.MenuItem;*

**Returns**

None.

**See Also**

The enable() and disable()methods of the MenuItem class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description.

## getLabel()

**ClassName**

MenuItem

**Purpose**

This method gets the label of this MenuItem.

**Syntax**

public String getLabel()

**Parameters**

None.

**Description**

Each menu item is associated with a string and can be uniquely identified by that string. This method gets the label, of type String, associated with this MenuItem. The label would have been set earlier either during construction or by the setLabel() method.

**Imports**

*import java.awt.MenuItem;*

**Returns**

The label identifying the MenuItem; return type is String.

**See Also**

The setLabel(String) method of the MenuItem class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. The label of the menu item ren is obtained using this method on the ren object.

## isEnabled()

**ClassName**

MenuItem

**Purpose**

This method checks to see if this MenuItem is enabled.

**Syntax**

public boolean isEnabled()

**Parameters**

None.

**Description**

A menu item is, by default, enabled on construction. Invoking the disable() method on the menu item disables any action on this item. You can use this method to check whether or not this MenuItem is enabled. The MenuItem is enabled and selectable by the user if this method returns true; if it returns false, this MenuItem is disabled and is not selectable by the user.

**Imports**

*import java.awt.MenuItem;*

**Returns**

A value of type boolean indicating whether this MenuItem is enabled or not.

**See Also**

The enable(boolean), enable(), and disable()methods of the MenuItem class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. This method is used to verify that the ren menu item is indeed disabled after the call to disable().

## paramString()

**ClassName**

MenuItem

**Purpose**

This method obtains the string parameter of this MenuItem.

**Syntax**

public String paramString()

**Parameters**

None.

**Description**

This method is used to obtain the String parameter of this MenuItem which contains the label of the menu item, apart from the other details of itself being a MenuComponent. This method overrides the paramString() method of the super-class MenuComponent. Also, note that this method is public, whereas the method paramString() of class MenuComponent is protected.

**Imports**

*package java.awt;*
*import java.awt.MenuItem;*

**Returns**

The String parameter of this MenuItem.

**Example**

The following code uses paramString by subclassing the MenuItem.

```
package java.awt;
import java.awt.MenuItem;

class myItem extends MenuItem {

       public myItem() {
        super();
       }

        public String getmyStringForm() {
                        return super.paramString();
        }

        public static void main(String[] args) {
              myItem item = new myItem();
              System.out.println(item.getmyStringForm());
        }
}
```

## setLabel(String)

**ClassName**

MenuItem

**Purpose**

This method gets the label of this MenuItem.

**Syntax**

public void setLabel(String label)

**Parameters**

***label***

The label to which this menu item has to be set.

**Description**

Each menu item is associated with a String label and can be identified by this String. This method sets the label, of type String, associated with this MenuItem. The label can be retrieved on any MenuItem using the getLabel() method.

**Imports**

*import java.awt.MenuItem;*

**Returns**

None.

**See Also**

The getLabel()method of the MenuItem class

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. The label of a menu item is changed from “Hi” to “Rename” using this method.

## MenuContainer

**Purpose**

Forms the super-class of all menu related container classes.

**Syntax**

public interface MenuContainer extends MenuComponent

**Description**

This is an interface in Java. A class that implements this interface effectively inherits this interface and hence this interface is said to form the super class of all menu related container classes. The classes Menu and MenuBar implement this interface and are container classes. This interface has three methods: getFont(), postEvent(Event), and remove(MenuComponent). Figure 6-15 illustrates the inheritance relationship of interface MenuContainer.

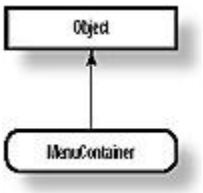


**Figure 6-15** Inheritance diagram of MenuContainer interface

**PackageName**

*java.awt*

**Imports**

*import java.awt.MenuContainer;*

**Example**

Refer to the usage of the methods of this interface in the class menu, given in the example description for class MenuComponent in Listing 6-2. MenuBar and Menu implement this interface and so these methods are invoked on instances of these classes.

## getFont()

**Interface**

MenuContainer

**Purpose**

This is an abstract method and has to be implemented by the class implementing this interface.

**Syntax**

public abstract Font getFont()

**Parameters**

None.

**Description**

This method has to be defined in the class that implements the MenuContainer interface. If it is set earlier using the setFont method, it returns the Font.

**Imports**

*import java.awt.MenuContainer;*

**Returns**

The font of the menu item; the return type is Font.

**See Also**

The Menu and MenuBar classes

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. After setting the font of the info menu, this method is used to print the details of the font to which it is set.

## postEvent(Event)

**Interface**

MenuContainer

**Purpose**

This is an abstract method and has to be implemented by the class implementing this interface.

**Syntax**

public abstract boolean postEvent(Event evt)

**Parameters**

***evt***

The event to be posted to a container object is passed as a parameter.

**Description**

This method has to be defined in the class that implements the MenuContainer interface. You can specify an event to be thrown on this MenuItem. The handleEvent() method should have appropriate action specified to handle this event to make usage of this method effective.

**Imports**

*import java.awt.MenuContainer;*

**Returns**

This method returns true if successful and false if not; return type is boolean.

**See Also**

The Menu and MenuBar classes

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. WINDOW_DESTROY event is posted using this method, which is implemented in class Menu.

## remove(MenuComponent)

**Interface**

MenuContainer

**Purpose**

This is an abstract method and has to be implemented by the class implementing this interface.

**Syntax**

public abstract void remove(MenuComponent mc)

**Parameters**

***mc***

The MenuComponent object that has to be removed from the container.

**Description**

This method has to be defined in the class that implements the MenuContainer interface.

**Imports**

*import java.awt.MenuContainer;*

**Returns**

None.

**See Also**

The Menu and MenuBar classes

**Example**

Refer to Listing 6-2, defining the class myMenu in MenuComponent class description. A menu tmp is created temporarily and is deleted from the menu bar using this method.

## Checkbox

**Purpose**

Encapsulates a user-interface element with a boolean state.

**Syntax**

public class Checkbox extends Component

**Description**

The Checkbox class encapsulates a user-interface element with boolean state. It is the radio button representation in Java. A Checkbox is either selected or not selected at any instant. When the user clicks on the Checkbox, its state toggles: if it was "on" it is reset to "off" and vice versa. A Checkbox item can belong to a CheckboxGroup, that is a group of Checkboxes. In this scenario, only one of the checkboxes can be "on" at any time. In case of X-Windows, it is selected in

depressed mode and if it is in raised mode, it is unselected. In Windows95, a check mark is placed next to the item if it is selected. You can specify to which group a checkbox will belong, either during construction or by using a method of this class. By default, a checkbox is initialized to a false state. Figure 6-16 illustrates the inheritance relationship of class Checkbox.

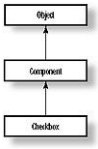


**Figure 6-16** Class diagram of the Checkbox class

**PackageName**

*java.awt*

**Imports**

*import java.awt.Checkbox;*

**Constructors**

public Checkbox()—constructor to create a Checkbox with no label, no CheckboxGroup, and initialized to false.
public Checkbox(String *label*)—constructor to create a Checkbox with a specified label, no CheckboxGroup and initialized to false.
public Checkbox(String *label*, CheckboxGroup *group*, boolean *state*) — constructor to create a Checkbox with a specified label, a specified CheckboxGroup and initialized to a specified state.

**Parameters**

***label***

The label, of type String, to be associated with the Checkbox.

***group***

The CheckboxGroup of which the Checkbox is made to be a member.

***state***

The specified state to which the Checkbox has to be initialized.

**Example**

The following code Listing 6-3 implements a class named check. Enter the code in a file named check.java and compile it. Executing the Java interpreter by typing java check will pop up the resultant window shown in Figure 6-17. This application prints either the string “hi” or the string “hello” at the center of the canvas. Two checkboxes, hi and hello, are provided. They form a group, so only one of them can be selected and, hence, printed on the canvas at a time. A single Checkbox is provided for the user to make the string print in red. If it is not selected, the string will be typed in blue. An OK button allows the user to approve the changes he makes.

**Figure 6-17** Resultant window after compiling and executing check.java

**Listing 6-3** check.java: Usage of checkboxes

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

class check extends Frame {
CheckboxGroup group;
Checkbox red;

public check() {

  setLayout(new FlowLayout());
  group = new CheckboxGroup();
  Checkbox hi = new Checkbox("hi”, group, true);
  Checkbox hello = new Checkbox("hello");
  hello.setCheckboxGroup(group);
  hello.setState(false);
  add(hi);
  add(hello);
  System.out.println(" Checkbox 'hello' belongs to the group " +
                   hello.getCheckboxGroup().toString());
  group.setCurrent(hi); // set hi to be default
  Button ok = new Button("OK");

  add(ok);

  red = new Checkbox("Red");
  add(red);
  red.setState(false);
  System.out.println(" Label of 'red' Checkbox is " + red.getLabel());

 }

public void paint(Graphics g) {

  //if (ok)
   //   ok = false; // set it to false

   if (red.getState())  // if red is selected
        g.setColor(Color.red);
   else
        g.setColor(Color.blue);

     g.setFont(new Font( "TimesRoman", Font.BOLD, 18));
   Checkbox cur = group.getCurrent(); // get the current selection:
   hi or hello
   String cur_str = cur.getLabel();
   if (cur_str.equals("hi"))
        g.drawString("hi", 150,100);
   else if (cur_str.equals("hello") )
        g.drawString("hello", 150,100);
   else g.drawString("None",150,100);

}

 public boolean action(Event evt, Object arg) {
```

```
  if (evt.target instanceof Button) {
        if ("OK".equals(arg)) {
      //      ok = true;
           repaint();
         }
 }
return true;

}

 public static void main(String args[]) {

        check chk = new check();
        chk.setTitle("Checkbox demo");

        chk.pack();
        chk.resize(300,200);
        chk.show();

 }

}
```

## addNotify()

**ClassName**

Checkbox

**Purpose**

This method creates a peer for the Checkbox. It lets you change the look of the Checkbox without changing its behavior.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

This method helps you change the look of the Checkbox without changing its behavior. It overrides the addNotify() method of class Component, and creates a peer for this Choice.

**Imports**

*import java.awt.Checkbox;*

**Returns**

None.

**See Also**

The *java.awt.peer;* CheckboxPeer interface; the Component class

## getCheckboxGroup()

**ClassName**

Checkbox

**Purpose**

This method returns the CheckboxGroup of which this Checkbox is a member.

**Syntax**

public CheckboxGroup getCheckboxGroup()

**Parameters**

None.

**Description**

A Checkbox can belong to a CheckboxGroup. This method returns the CheckboxGroup to which this Checkbox belongs, if it's a part of a CheckboxGroup.

**Imports**

*import java.awt.Checkbox;*

**Returns**

None.

**See Also**

The CheckboxGroup class; the setCheckboxGroupmethod of the Checkbox class

**Example**

Refer to the example in Listing 6-3. The CheckboxGroup of the hello Checkbox is obtained in the example using this method.

## getLabel()

**ClassName**

Checkbox

**Purpose**

This method obtains the label of this Checkbox button.

**Syntax**

public String getLabel()

**Parameters**

None.

**Description**

A Checkbox can be identified by its label. This method is especially helpful when you are trying to handle an event on a Checkbox and you want to find out which checkbox button the user has clicked on. This method returns the label of this Checkbox as a String.

**Imports**

*import java.awt.Checkbox;*

**Returns**

The label of this Checkbox; the return type is String

**See Also**

The setLabel method of the Checkbox class

**Example**

Refer to the example in Listing 6-3. This method is used to obtain the label and find out whether it is a "hi" or "hello" checkbox.

## getState()

**ClassName**

Checkbox

**Purpose**

This method returns the boolean state of this Checkbox, indicating whether it is selected or not.

**Syntax**

public boolean getState()

**Parameters**

None.

**Description**

A Checkbox toggles between an "on" and "off" state on every mouse click on the Checkbox. This method returns the state of the Checkbox at the time of invocation.

**Imports**

*import java.awt.Checkbox;*

**Returns**

A boolean value indicating the state of the checkbox button.

**See Also**

The setState method of the CheckboxGroup class

**Example**

Refer to the example in Listing 6-3. This method is used to find whether "red" or "blue" is selected by finding the state of the checkbox.

## paramString()

**ClassName**

Checkbox

**Purpose**

To obtain the parameter String of this Checkbox.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method returns the parameter String of this Checkbox which is, effectively, the label associated with this Checkbox and the state of the checkbox button.

**Imports**

*import java.awt.Checkbox;*

**Returns**

Parameter string of type String, of this Checkbox.

**See Also**

The getLabelmethod of the Checkbox class

**Example**

The following code uses paramString by subclassing the Checkbox.

```
package java.awt;
import java.awt.Checkbox;
```

```
class myBox extends Checkbox {

        public myBox(String boxname) {

        }

        public String getmyStringForm() {
                        return super.paramString();

        }

        public static void main(String[] args) {
               myBox box = new myBox("TestBox");
               System.out.println(box.getmyStringForm());
         }
}
```

## setCheckboxGroup(*CheckboxGroup*)

**ClassName**

Checkbox

**Purpose**

This method sets a Checkbox to belong to the specified CheckboxGroup.

**Syntax**

public void setCheckboxGroup(CheckboxGroup cb_grp)

**Parameters**

***cb_grp***

CheckboxGroup to which this Checkbox is added

**Description**

A Checkbox can belong to a CheckboxGroup. This method sets the group to which this Checkbox will belong. Only one Checkbox can be "on" at any time among all the Checkboxes belonging to a group. This formation is helpful when you are using a CheckboxGroup in your interface to select among a set of options.

**Imports**

*import java.awt.Checkbox;*

**Returns**

None.

**See Also**

The CheckboxGroup class; the getCheckboxGroupmethod of the Checkbox class

**Example**

Refer to the example in Listing 6-3. This method is used to set the group of the "hello" checkbox to the same group as the "hi" checkbox.

## setLabel(String)

**ClassName**

Checkbox

**Purpose**

The label of this Checkbox button is set to the specified string value.

**Syntax**

public void setLabel(String label)

**Parameters**

***label***

The String type value that forms the label of this button.

**Description**

A Checkbox can be identified by its label. This method can be used when you have created a Checkbox, without specifying a label during construction, because the value might be available only at a later stage and not during construction.

**Imports**

*import java.awt.Checkbox;*

**Returns**

None.

**See Also**

The getLabel method of the Checkbox class

**Example**

Refer to the example in Checkbox class description, Listing 6-3.

## setState(*boolean*)

**ClassName**

Checkbox

**Purpose**

Sets the boolean state of this Checkbox to the specified state.

**Syntax**

public void setState(boolean state)

**Parameters**

***state***

The state, of type boolean, that indicates whether to set this Checkbox to be enabled or disabled.

**Description**

A Checkbox toggles between an "on" and "off" state on every mouse click on the Checkbox. This method sets the state of the Checkbox to the specified boolean value. It can be useful when you are initializing your interface environment and setting some of checkbox values to true and others to false. Also you might want to set the state of a checkbox button to reflect the occurrence of an event. When the button is set to true, it will appear in depressed position, and when the value is false, it will appear in raised position.

**Imports**

*import java.awt.Checkbox;*

**Returns**

None.

**See Also**

The getState() method of the CheckboxGroup class

**Example**

Refer to the example in Checkbox class description, Listing 6-3.

## CheckboxGroup

**Purpose**

Helps to create a group of mutually exclusive Checkbox items such that only one item can be "on" at a time.

**Syntax**

public class CheckboxGroup extends Object

**Description**

A set of Checkbox buttons can be grouped together to form a CheckboxGroup. These buttons are grouped such that only one of them can be "on" at a time. Clicking on any of the Checkbox buttons in a group, enables that Checkbox and all other Checkboxes belonging to that group are automatically disabled, which is also reflected in the appearance of the buttons. This way, you can form a set of options in an interface, where a user can select only one of the options. You can specify which group a Checkbox will belong to, either during its construction or by using a method of Checkbox class. By default, all Checkboxes are initialized to false. Figure 6-18 illustrates the inheritance relationship of class CheckboxGroup.

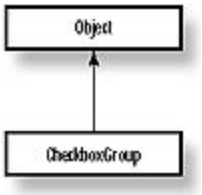


**Figure 6-18** Class diagram of the CheckboxGroup class

**PackageName**

*java.awt*

**Imports**

*import java.awt.CheckboxGroup;*

**Constructors**

public CheckboxGroup()

**Parameters**

None.

**Example**

Refer to the example check.java.

## getCurrent()

**ClassName**

CheckboxGroup

**Purpose**

This method obtains the currently selected Checkbox among the set of Checkboxes in this group.

**Syntax**

public Checkbox getCurrent()

**Parameters**

None.

**Description**

In a user-interface, you can find out which Checkbox button the user has selected by examining the state of all Checkboxes. When Checkboxes are grouped, they are mutually exclusive. Only one of the Checkboxes can be "on" at a given time. With this knowledge, it is enough if you know which is the current choice in the group. With this method, you can obtain the necessary information about the current choice in a CheckboxGroup without examining each Checkbox. This method finds the currently selected Checkbox and returns a reference to that object.

**Imports**

*import java.awt.CheckboxGroup;*

**Returns**

A reference of type Checkbox to the currently selected Checkbox item in the group.

**See Also**

The setCurrent method of the CheckboxGroup class

**Example**

Refer to the example Listing 6-3 in Checkbox class description. The current selection among the "hi" and "hello" checkboxes is found using this method.

## setCurrent(*Checkbox*)

**ClassName**

CheckboxGroup

**Purpose**

This method sets the specified Checkbox as the current choice among all Checkbox items in this group.

**Syntax**

public void setCurrent(Checkbox curr_cb)

**Parameters**

***curr_cb***

A reference to the Checkbox item which is to be set as the current choice.

**Description**

A Checkbox gets selected in a group when the user clicks on the Checkbox button. If you have information that indirectly selects a Checkbox or if you want to explicitly specify the current choice Checkbox in a group, you can use this method. This method sets the specified Checkbox to "on," and other Checkboxes in the same group are automatically disabled.

**Imports**

*import java.awt.CheckboxGroup;*

**Returns**

None.

**See Also**

The getCurrent method of the CheckboxGroup class

**Example**

Refer to the example in Checkbox class description, Listing 6-3. The "hi" checkbox is set to be the default selected box, using this method.

## toString()

**ClassName**

CheckboxGroup

**Purpose**

This method obtains the string representation of the values in this CheckboxGroup.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

A CheckboxGroup contains two or more Checkboxes. Each Checkbox has a unique label associated with it. This method returns the string form of all the values of the Checkboxes that are part of this CheckboxGroup.

**Imports**

*import java.awt.CheckboxGroup;*

**Returns**

Values, of type String, representing the values of this CheckboxGroup.

**Example**

Refer to the example in Checkbox class description, Listing 6-3.

## CheckboxMenuItem

**Purpose**

Produces a Checkbox to represent a choice in a menu.

**Syntax**

public class CheckboxMenuItem extends MenuItem

**Description**

A Menu consists of a set of menu items. This class allows you to specify a checkbox as a menu item. Bacause it subclasses the MenuItem class, an instance of this class can be a part of a Menu. Since it forms part of the menu, you must specify a label for the item. Figure 6-19 illustrates the inheritance relationship of class CheckboxMenuItem.

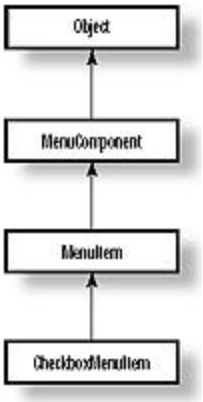


**Figure 6-19** Class diagram of the CheckboxMenuItem class

**PackageName**

*java.awt*

**Imports**

*import java.awt.CheckboxMenuItem;*

**Constructors**

public CheckboxMenuItem(String *label*)

**Parameters**

***label***

The String value of this menu item which identifies it among other items in a menu.

**Example**

Refer to the Selections Interface application built at the end of this chapter.

## addNotify()

**ClassName**

CheckboxMenuItem

**Purpose**

This method creates a peer for this CheckboxMenuItem. It helps you change the look of the Checkbox without changing its behavior.

**Syntax**

public synchronized void addNotify()

**Parameters**

None.

**Description**

This method helps you change the look of the Checkbox menu item without changing its behavior. It creates a peer for this CheckboxMenuItem, and overrides the addNotify() method of class MenuItem.

**Imports**

*import java.awt.CheckboxMenuItem;*

**Returns**

None.

**See Also**

The java.awt.peer; the CheckboxMenuItemPeer interface

**Example**

Refer to Chapter 9 on peers and interfaces.

## getState()

**ClassName**

CheckboxMenuItem

**Purpose**

This method returns the boolean state of this CheckboxMenuItem indicating whether it is selected or not.

**Syntax**

public boolean getState()

**Parameters**

None.

**Description**

A Checkbox toggles between an "on" and "off" state on every mouse click on the Checkbox. This method returns the state of the CheckboxMenuItem at the time of invocation. It can be invoked on an instance of menu item which is also a Checkbox. When the button is depressed or has a check mark next to it, it is "on" and it is "off" when it is in raised position or when no check mark is placed next to it.

**Imports**

*import java.awt.CheckboxMenuItem;*

**Returns**

A boolean value, indicating the state of the checkbox button which is apart of a Menu.

**See Also**

The setState method of the CheckboxMenuItem class

## paramString()

**ClassName**

CheckboxMenuItem

**Purpose**

This method obtains the parameter String of this CheckboxMenuItem button.

**Syntax**

protected String paramString()

**Parameters**

None.

**Description**

This method returns the parameter String of this CheckboxMenuItem which is, effectively, the label associated with this checkbox button in a menu and the state of the checkbox menu item.

**Imports**

*import java.awt.CheckboxMenuItem;*

**Returns**

A parameter string of type String which applies to this checkbox item.

**See Also**

The getLabel method of the Checkbox class

**Example**

The following code uses paramString by subclassing the CheckboxMenuItem.

```
package java.awt;
import java.awt.CheckboxMenuItem;

class myCItem extends CheckboxMenuItem{

       public myCItem(String lbl) {
        super(lbl);0
       }

        public String getmyStringForm() {
                        return super.paramString();
        }

        public static void main(String[] args) {
              myCItem item = new myCItem("TestItem");
              System.out.println(item.getmyStringForm());
        }
}
```

## setState(*boolean*)

**ClassName**

CheckboxMenuItem

**Purpose**

This method sets the boolean state of this CheckboxMenuItem to the specified state.

**Syntax**

public void setState(boolean state)

**Parameters**

***state***

The state of type boolean that indicates whether to set this item to enabled or disabled.

**Description**

A Checkbox toggles between an "on" and "off" state on every mouse click on the Checkbox. This property holds true even when it is a menu item and forms a part of a menu. This method sets the state of the CheckboxMenuItem to the specified boolean value.

**Imports**

*import java.awt.CheckboxMenuItem;*

**Returns**

None.

**See Also**

The getState method of the CheckboxMenuItem class

## The Selections Interface Application

Now you're ready to develop a user-interface application that uses various GUI components to offer options to the user. The Selections Interface Application is a Java stand-alone user-interface that allows a user to enter a text string and then choose the size and color in which the string will be displayed. There's also an option to use bold style for the text, and even one to make the text stand still or dance in the middle of the canvas. There are many ways to implement this application. In this project, you will follow one method all the way through. Here are the components you will be creating:

**1.** A text field for the user to enter a string.
**2.** A menu named Color from which the user can choose red, green, and blue.
**3.** A tear-off menu titled Quit and that contains Exit as an item.
**4.** A menu titled Help, that appears at the top right of this application and contains a menu item labeled Info.
**5.** A checkbox to the left of the text field that allows the user to display the text in bold.
**6.** A Choice button to the right of the text field that allows users to specify the text font size as 12, 18, 24, or 26 point.
**7.** You shall place the checkbox, text field, and choice at the top of your window canvas.
**8.** Two radio buttons labeled Simple and Fun. When Simple is selected, the text string will stand still at the center of the canvas. Since the user should be able to select only one of these two buttons at any time, we'll use the CheckboxGroup class to create them.
**9.** A button labeled OK, to the left of the radio buttons, which the user must press to confirm any selection. Only when the user presses this button, will any selection take effect. Until the user presses OK, the earlier selection will remain displayed.

### Building Your Application

**1.** First create a Selections class. As the interface should contain a menu bar according to the specification, your Selections class should subclass the Frame. You can attach a menu bar only to a Frame object using the setMenuBar() method of Frame. Also, because we're providing the functionality to make the text dance, you need a Thread, running in a loop, to choreograph the text's movements; hence, the class should implement the Runnable interface. Enter the following code in a file named Selections.java. The class should be public.

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

```
/**

        Filename: Selections.java

        Classname: Selections

        Purpose:   An application to illustrate the usage of GUI
for
        making selections, providing options in Java. Usage of
Menus,
        Checkboxes, radio buttons, Choice menu in Java's
        AWT classes.

**/

public class Selections extends Frame implements Runnable {

}
```

**2.** To add a text field in your interface, you'll create a member in the class of type TextField and instantiate this member within the constructor for this class. Make the initial string in the text field "Hello" and let ten be the number of columns in the field. Add the following lines in the Selections class.

```
TextField txt_fld;

public Selection() {

/* TextField to enter any string */
txt_fld = new TextField("Hello", 10);
}
```

**3.** Now let's add the Color menu. You should have three colors as items in this menu. Let a variable, txt_color, be a member in the class so that the selected color is accessible within this class. Initialize this variable to black so that, by default, the text will be displayed in black. Construct the non-tear-off menu with title Color. Enter the following lines inside the class, but outside the constructor.

```
Color txt_color = Color.black; // add this outside the
constructor

/* Enter the following lines inside the constructor Selections()
to construct the Color menu.
 Provide a menu to choose a color to display the text in */
        Menu col_menu = new Menu("Color", false);
                // not a tear-off menu
        col_menu.add(new MenuItem("Red"));
                // add "Red" as a menu item
        col_menu.add("Green");
                // add Green and blue as labels
        col_menu.add("Blue");
```

**4.** Enter the following code in the constructor. This will create the Quit menu with Exit as a CheckboxMenuItem and will also create a menu titled Help containing one item labeled Info.

```
   Menu quit_menu = new Menu("Quit");
   quit_menu.add(new CheckboxMenuItem("Exit"));

   Menu help_menu = new Menu("Help");
```

```
   help_menu.add("Info"); //nothing implemented - just for demo
```

**5.** Having created the menus, you can create a menu bar to contain all these. Then the menu bar should be added to the Frame. As Selections is a subclass of Frame, you can call the setMenuBar() method directly. Position the Help menu on the rightmost side of the menu bar using the setHelpMenu method. Include the following lines in the constructor.

```
/* Add the menu to the menu bar at the top of the frame */
        MenuBar mybar = new MenuBar();
        mybar.add(quit_menu);
        Menu ret = mybar.add(col_menu);
        mybar.add(help_menu);
        mybar.setHelpMenu(help_menu);
        System.out.println(" addmenu returns -> " +
ret.toString());
        setMenuBar(mybar);
```

**6.** You can now create a Choice button for selecting the font size. Add 12, 18, 24, and 26 to the Choice as options for font size. Create a member bold_box of type Checkbox in class Selections. Initialize this checkbox in the constructor. Add the following line outside the constructor.

```
   Checkbox bold_box;
```

**7.** Now include the following code inside the constructor where the txt_fld is initialized so that the Checkbox, text field, and Choice appear at the top of the canvas in a single row. You're adding these components to a new panel and appending the panel to the "North" of the main panel, the frame.

```
   /* A Checkbox to toggle the text between PLAIN and BOLD styles
*/
   bold_box = new Checkbox("Bold");

   /* TextField to enter any string */
   txt_fld = new TextField("Hello",10);

   /* A Choice menu to provide a list of size options for the
text */
   Choice fontchoice = new Choice();
   fontchoice.addItem("12");
   fontchoice.addItem("18");
   fontchoice.addItem("24");
   fontchoice.addItem("26");

   /* Add the Checkbox, TextField, and Choice to a panel */
   Panel tpanel = new Panel();
   tpanel.setLayout(new FlowLayout());
   tpanel.add(bold_box);
   tpanel.add(txt_fld);
   tpanel.add(fontchoice);

            add("North", tpanel);
```

**8.** All that's left in the interface to create are the OK button and the two radio buttons at the bottom of the main frame. Create a panel and add the OK button to it. Also, create two Checkboxes to indicate the selection between Simple and Fun. Form a CheckboxGroup with these two checkboxes so that only one of them can be selected by the user. Add this to a panel placed at the "South" of the main frame. The following code achieves this effect.

```
   /* A button to confirm any action to be taken */
   Panel p_bot = new Panel(); // panel to attach to the bottom
   p_bot.add(new Button("OK"));

   /* two checkboxes */
   myradiobox = new CheckboxGroup();
   p_bot.add(new Checkbox("Fun", myradiobox, false));
   p_bot.add(new Checkbox("Simple", myradiobox, true));

   add("South", p_bot);
```

**9.** Create a Thread and pass the CheckboxGroup from the Selection object as the target for the Thread. Call this method from the Selections constructor. Define the method run() as the Selections class implements the interface Runnable. While the Thread exists, pause it so that the CPU becomes idle and executes the paint method. Add the following code inside class Selections.

```
/* A thread to implement jumping text */
        Thread winThread = null;

        /* Method winStart() to start the thread of this class */
        private void winStart() {
        if(winThread == null)
          winThread = new Thread(this);
          winThread.start();
        }

        /* Method run() of interface Runnable defined here */
        public void run() {
         while(winThread !=null) {
                try{
                 Thread.sleep(100);
                } catch (InterruptedException e){}
                        repaint();
                 }
        }
```

**10.** Having constructed the interface, you must now handle the events that will be generated when the user clicks on the OK button or when the user makes a selection from the Color menu. When the Exit menu item is selected by the user, you should exit from the application. These events can be handled by using the action() method. Enter the following code in the Selections class.

```
boolean ok = false;   // a member to track if ok is
pressed by the user

/* Method to handle Events in this GUI
  Action is taken if "OK" button is pressed,
                if Menu is chosen
                  Choice is selected
*/
public boolean action(Event evt, Object arg) {
 if(evt.target instanceof Button) {
     if("OK".equals(arg)) {
         ok = true;
         repaint();
       }

     }
```

```
     else if(evt.target instanceof Choice) {
             Choice c = (Choice)(evt.target);
             font = c.getSelectedItem();
     }
     else if(evt.target instanceof MenuItem) {
            /* Make changes to reflect the selection of one color
              Make other colors false and selected color true
            */
            // menu item is selected; so get the menu in which
this
            item is a member

          Menu menu = (Menu)evt.target.getParent();
          if (menu == col_menu) {
          if ("Red".equals(arg)) {
                 txt_color = Color.red;
          }
          if ("Green".equals(arg)) {
                txt_color = Color.green;
           }
          if ("Blue".equals(arg)) {
                 txt_color = Color.blue;
          }
          }
          else if (menu==quit_menu)
          if("Exit".equals(arg)) {
                System.exit(0);
          }
     }

     return true;
}
```

**11.** When OK is pressed, note that we are making a call to the method repaint(). This, in turn, will schedule a call to the paint() method with the Graphics object under current context. You should change the paint() method to implement any graphical changes in the interface. In the paint() method, we make changes only if the user presses the OK button. (Note: Even when the OK button is not pressed, repaint() is called in the run() method ). Check the checkbox for Bold style. If it is enabled, the text should be in bold. Find out which of the Checkboxes in the CheckBox group is selected, i.e., Simple or Fun type. Obtain the text string from the TextField. Then either display it on the canvas or make it dance, depending on the user's selection task. These are tasks performed in the paint() method defined here. Add this method to the Selections class.

```
public void paint(Graphics g) {

   if(ok) {
      g.setColor(txt_color);    // set the color to the selected
color
       ok = false; // make it true ONLY when the user clicks on
it.
       Font myfont;
       // set the style to BOLD or PLAIN depending on bold
checkbox
       selection
```

```
      if (!bold_box.getState())
     myfont = new Font(font, Font.PLAIN,
     Integer.valueOf(font).intValue());
      else
     myfont = new Font(font, Font.BOLD,
     Integer.valueOf(font).intValue());
   g.setFont(myfont);

   /* Which of the checkboxes is selected?
       To display the text statically or as jumping text?
    */

   String selctn =  (myradiobox.getCurrent()).getLabel();

   /* obtain the text entered in the TextField */
   String str = txt_fld.getText().trim();
   if(selctn.equals("Simple"))
        g.drawString(str,200,150);
   else if(selctn.equals("Fun")) {

        char str_chars[];
        /* get the characters in the string and display them
        one by one at random co-ord */
          str_chars =  new char [str.length()];

        str.getChars(0,str.length(),str_chars,0);
          for(int i=0;i<str.length();i++)

        {


   int x_coord = (int) (Math.random()*3+10*i+200);

   int y_coord = (int) (Math.random()*10+150);
                g.drawChars(str_chars, i,1,x_coord,y_coord);

        }
   }

    } // end of " if (ok) "
  }    // end of paint() method
```

**12.Now** that you've implemented all these methods, all that is required to complete this application is to make it stand-alone. Include the following main() method in the class Selections. This constructs an object of type Selections and displays it, which forms our required user-interface for providing selections to the user. Figure 6-20 captures the Selection interface in action.

```
public static void main(String args[]) throws IOException {
   Selections mywin = new Selections();
   mywin.setTitle("Selections Applet");
   mywin.pack();
   mywin.resize(400,300);
         mywin.show();
}
```

**Figure 6-20** The Selections Interface application in action

### How It Works

The Selections Interface application illustrates the use of windowing components that provide the funtionality of implementing selection and confirmation in user-interface applications. When the application starts, the window contains a text field to enter a string. The default string that appears in the text field is “Hello”. There is a checkbox button to the left of the text field to specify the string to be displayed in bold font. The choice menu button on the right side of the text field is for selecting the font size of the text to be displayed. There are three menus provided as a part of the menu bar. The menu titled Color is provided to select the color in which the text is to be displayed. The menu titled Quit is to be used to exit from the application and the menu titled Help is placed on the rightmost side of the menu bar.

There are two checkbox buttons labelled Fun and Simple. If the Simple button is selected, the string will be displayed on the window. If the Fun button is selected, the displayed string will start dancing in the window. These two buttons form a mutually exclusive group, i.e., only one of them can be selected at any given instant. Try changing the string in the text field and start playing around by varying all these parameters (size, font, color, Fun, or Simple). Have fun

# Chapter 7
# Color, Font, Images, And Shapes

The Java API includes several classes designed to make Graphics rendering operations easier. Classes exist to represent colors, fonts, shapes, and images. Instances of these classes are passed, as parameters, to Graphics objects to facilitate rendering on a display surface.

Without these classes, specifying colors, fonts or simple shapes would require passing multiple parameters to Graphics methods; instead you can simply instantiate an instance of a particular graphical helper class and pass it to the Graphics object. For example, the Color class represents a color as three component color member variables: red, green and blue. Without this class, specifying a color to a Graphics object for rendering some geometric primitive, such as a line or oval, would require three parameters, one each for the red, green and blue color components.

The Graphics method for specifying the foreground color is Graphics.setColor(). The signature of setColor() is

```
public abstract class Graphics {
    public void setColor( Color c );
}
```

Without the Color class, this single parameter would be replaced by three, making coding more complex:

```
public abstract class Graphics {
    public void setColor(int r, int g, int b);
}
```

In addition to encapsulating the data, the graphical helper classes also provide methods for common tasks associated with the data. For example, the Polygon class includes a getBoundingBox method which calculates the smallest rectangle that completely contains the polygon. As the project for this chapter, Doodle, demonstrates, Polygon.getBoundingBox comes in quite handy at times when you are working with Polygons. In addition, the implementations of common tasks in graphical helper class methods often cuts down on code size significantly.

The project for this chapter demonstrates use of the graphical helper classes by implementing a relatively simple Doodle application. In addition, the Doodle application demonstrates the use of peer-less Components, which I call "virtual Components," to manage overlapping rectangular areas of a window.

## Colors

In Java, the default way to describe colors is by using the very common RGB color model. Color class instances assume colors will be presented that way. For those unfamiliar with this method of describing colors, here's a quick summary: Every color is described by a combination of red, green, and blue color components. Each component is given an absolute magnitude, from zero to some maximum. In Java, each color component is storable in an unsigned byte, and thus, is in the rangle 0-255. Each unique color is a unique point in the three-dimensional space illustrated in Figure 7-1. For example, absolute red is the point (255,0,0) in Java. The color white is represented as (255,255,255). And black is represented (0,0,0). (I was taught from a wee age that "black is not a color." If you, too, were subjected to this philosophy, you may have to realign a handful of synapses to deal with the fact that black is representable in the RGB color model, and so, is indeed a "color" for the purposes of this chapter.)

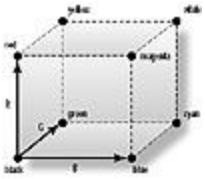

**Figure 7-1** The RGB bounded color space

Java API's Color class has two ways of representing RGB color components to or from a Color object. Each of the three color components can be passed in a separate integer, as in the Color class constructor

```
public Color(int red, int green, int blue);
```

Alternatively, all three color components can be packed into a single integer. Since each color component may have only the values 0-255, each can be packed into a distinct byte of a four-byte integer. Packed RGB components are used in the Color methods, as shown here

```
public Color(int rgb);  // Constructor taking packed RGB data
public int getRGB();    // Returns Color's packed components
```

When RGB color components are packed this way, the resultant integer is formulated like this: 0xFFrrggbb. That is, the top byte contains 0xFF. (Actually, it may contain any value. This value is ignored.) The next byte holds the red color component. The next holds green, and the bottom byte holds the blue color component. This listing shows how to build a packed integer of RGB color components from three distinct color component integer variables.

```
public int packComponents(int red, int green, int blue) {
    int colorRet = 0xFF000000 |
                   (red << 16) |
                   (green << 8) |
                   blue;
    return colorRet;
}
```

The Color class provides two RGB constructors, one of which is passed three distinct color component integer parameters for RGB values. The other RGB constructor takes a single packed integer of the color component data.

Color class instances also expose their RGB values through two types of methods. The getXXXmethods getRed, getGreen,and getBlue, each return a single integer indicating the Color's value for a single RGB color component. The getRGB method, as mentioned above, returns a packed integer holding all three color component values.

You may notice the similarities between the Color class packed component color format and the ColorModel packed RGBA format for representing colors in an Image. Although the formats are very similar, the chief difference is that the ColorModel has a fourth color

component, “translucency” or “A,”, stored in the top byte of the packed integer. Java Color objects do not deal with translucency values, and thus ignore the values in the top byte of the packed RGB data

## The HSB Color Model

Next to the RGB color model, the most commonly understood color model is the so-called HSB color model. The Color class provides some methods for automatically converting color component data between RGB and HSB color models. Figure 7-2 illustrates the HSB color space.

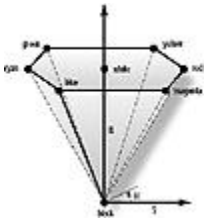

**Figure 7-2** The HSB color space

HSB color model methods are provided in the Color class because some color based operations are better carried out using HSB data. For example, converting a Color to grayscale using HSB data involves only zeroing-out the S color component. This is a much harder operation in the RGB color model. On the other hand, it is generally much easier to recognize the RGB triplet (0,255,0) as the color green, as opposed to the HSB triplet (1,1,2π/3) which also represents green.

For readers unfamiliar with the HSB color model, here’s a quick summary. The HSB color model breaks colors into three color components: “brightness,” “saturation,” and “hue.” Brightness is easiest to understand as the grayscale value of a particular color. Saturation describes how bold the color is. Less saturated colors are more washed out. More saturated colors are more vibrant. Note that a color with 0 saturation is actually a grayscale color. Finally, hue indicates a point on the standard color wheel that is closest to a particular color.

Generally, HSB values are described on a scale of 0-1 for the brightness (B) and saturation (S) values. The hue is an angle around the central axis of the HSB color space (or around the standard color wheel, however you want to look at it). Hue values in Java are given in radians. All three values are passed as float valiables in the Color class.

The static Color class methods RGBtoHSB and HSBtoRGB provide a simple method for converting between the RGB and HSB color models. In addition, getHSBColor is a static method that will create a new Color object given the HSB color components. An HSB color model constructor is not provided by this class, probably to avoid coding confusion since it would be diffucult at first glance to tell whether a particular constructor call was invoking the RGB or HSB color model overloaded constructor version.

## Using Colors

Color objects are generally used in AWT methods to refer to a Component object's foreground or background colors. Component.setForeground and setBackground each take a single Color object parameter. Similarly, Component.getForeground and getBackground return a Color object, representing the colors that the Component uses to draw in its update and paint methods.

To set the color a Graphics object uses when drawing to its display surface, use Graphics.setColor, passing it a Color object representing the color you want to use. The getColor method of the Graphics class returns the color it will use when drawing.

Note that several public, static Color object instances exist in the Color class. Each instance represents a common color, such as red, magenta, cyan, or white. These Color instances can be used as a shorthand instead of creating new Color objects to represent these common colors. The following listing shows how using these static instances can make coding a little bit easier: The class Palette presents a series of radio buttons. Each button uses a different background color. Such a panel is the basis for a decent "color chooser" toolbar for use in any application where the user chooses colors. Figure 7-3 shows an Applet that uses this Palette implementation.

**Figure 7-3** The Palette panel allows users to select from one of the static Color members of the Color class

```
ublic class Palette extends Panel {
    public Palette() {
        // Radio buttons presented in a series of centered
        // horizontal rows, controlled by a FlowLayout layout
        // manager.
        setLayout(new FlowLayout(FlowLayout,CENTER, 0, 0));

        // Add a radio button for each of the static Color object
        // instances which are members of the Color class.
        CheckboxGroup group = new CheckboxGroup();
        Checkbox white = new Checkbox("", group, true);
        Checkbox lightGray = new Checkbox( "", group, false);
        Checkbox gray = new Checkbox( "", group, false);
        Checkbox darkGray = new Checkbox("", group, false);
        Checkbox black = new Checkbox("", group, false);
        Checkbox red = new Checkbox("", group, false);
        Checkbox pink = new Checkbox("", group, false);
        Checkbox orange = new Checkbox("", group, false);
        Checkbox yellow = new Checkbox("", group, false);
        Checkbox green = new Checkbox("", group, false);
        Checkbox magenta = new Checkbox("", group, false);
        Checkbox cyan = new Checkbox("", group, false);
        Checkbox blue = new Checkbox("", group, false);

        // Set background colors for each radio button using
        // corresponding static Color object instance.
        white.setBackground( Color.white );
```

```
        lightGary.setBackground( Color.lightGray );
        gray.setBackground( Color.gray );
        darkGray.setBackground( Color.darkGray );
        black.setColor( Color.black );
        red.setBackground( Color.red );
        pink.setBackground( Color.pink );
        orange.setBackground( Color.orange );
        yellow.setBackground( Color.yellow );
        green.setBackground( Color.green );
        magenta.setBackground( Color.magenta );
        cyan.setBackground( Color.cyan );
        blue.setBackground( Color.blue );

        // Add all of the radio buttons to this panel.
        add(white);
        add(lightGray);
        add(gray);
        add(darkGray);
        add(black);
        add(red);
        add(pink);
        add(orange);
        add(yellow);
        add(green);
        add(magenta);
        add(cyan);
        add(blue);
    }
}
```

## Fonts: The Facts About Rendering Text

Font objects represent individual typefaces and styles used to render text on a drawing surface. Fonts are specified by a typeface name, and additional style and size indicators describe variations of the typeface. The Font class constructor is used to instantiate Font objects.

```
Font myFont = new Font("Helvetica", nStyleFlags, nPointSize);
```

The set of available typefaces is system-dependent. That is, font typefaces available on Windows 95 systems are not necessarily the same as those available on a Solaris or Macintosh. To get the list of available typeface names on any system, you must use the Toolkit's getFontList method. The getFontList method returns an array of String objects, each element of the array containing one typeface name available for AWT Graphics text rendering.

```
String[ ] astrFontList = Toolkit.getDefaultToolkit().getFontList();
for( int ii=0 ; ii<astrFontList.length ; ii++ ) {
    System.out.println( astrFontList[ii] );
}
```

The typeface can be modified by style attributes. There are three style attributes defined for all typefaces. The following Font class static members represent these style flags.

| Typeface Style Flag | Description |
|---|---|
| Font.PLAIN | Unmodified typeface. The constant Font.PLAIN is defined as 0. |
| Font.ITALICS | Italicized version of the typeface. |
| Font.BOLD | Bold version of the typeface. |

The Font's style is a bitwise ORing of these flags. Later versions of the Java API will, undoubtedly, include more Font style flags. For example, to create a Font object using both the bold and italics styles, you would use the Font constructor like this:

```
Font myFontObject = new Font("Helvetica", FGont.BOLD |
Font.ITALICS, nPointSize);
```

The size of a font is specified in "points", which is a typographic unit equal to (just about) 1/72 inch. This distance specifies the distance from the bottom of descending characters to the top of ascending characters of the typeface. Figure 7-4 illustrates the various metrics associated with a typeface, including the height, or point size, of the Font which is passed as the third parameter of the Font class constructor.


**Figure 7-4** Typographical metrics that describe a font

### Measuring a Font: The FontMetrics Class

The font metrics illustrated in Figure 7-4 are not directly available from a given Font object. Instead, you get a FontMetrics for a given Font object. The FontMetrics provides these metrics through its public class methods. The following lists the metrics of a Font available through the FontMetrics class, and the methods that provide access to those metrics.

| Metric | Methods and Description of the Metric |
|---|---|
| Leading | The suggested distance between successive lines of rendered text. This is the distance between the bottom of the descending characters of the previous line and the top of the ascending characters of the next line. Another way to say this is that the distance between baselines of successive lines of text should be the Leading + Ascent + Descent. FontMetrics.getLeading() returns the Leading of the associated Font. |

| | |
|---|---|
| Ascent | The distance from the baseline to the top of asending chartacters. FontMetrics.getAscent() returns the Ascent of a given Font. |
| Descent | The distance from the baseline to the bottom of descending characters. FontMetrics.getDescent() returns the Descent of a given Font. |
| Height | The distance between baselines of successive lines of text, which is determined as Leading + Ascent + Descent. This is not the same thing as the font's typesize, which is simply Ascent + Descent, or the distance from the top of ascending characters to the bottom of descending characters. FontMetrics.getHeight() returns the Height of a given font. |
| Width | Fixed-width fonts have the same width for all characters. Variable-width fonts, such as the font used to display this sentence, have characters of different widths. The FontMetrics class includes methods to measure the width of a particular character, or of a string of characters. These methods are charWidth(), stringWidth(), charsWidths(), bytesWidth(). |

---

Why have a FontMetrics class at all? Why not have the Font object provide methods to access a font's metrics? The reason is, the same font may have different metrics when used to render text on different display surfaces. For example, a character of 10-point Helvetica font displayed on the screen may have a different actual size if displayed on a printer. If the printer must use an alternative font because it does not know the Helvetica typeface, then the sizes will most certainly be different. The FontMetrics class represents the metrics of a particular Font when used on a particular display surface.

The FontMetrics constructor accepts a Font object as its only parameter. The FontMetrics object which is created contains the metrics of the Font when used to render text on the default display surface (usually the screen), as shown here:

```
Font f = new Font("Helvetica", Font.ITALICS | Font.BOLD, 10);
FontMetrics fm = new FontMetrics(f);

// fm contains metrics of Font f when used to display text on
// the screen.
```

To get the metrics for text of a particular font when displayed on another display surface, you must use a Graphics object attached to that display surface. These are the steps:

1. Get a Graphics object associated with the alternative display surface.
2. Associate the Font you want to measure with the Graphics object using setFont.

**3.** Get the FontMetrics for the Font when used to display text on the display surface using getFontMetrics. Here's how it looks:

```
Graphics g;

// Instantiate Graphics g by associating with display surface.

g.setFont(myFont);
FontMetrics fm = g.getFontMetrics();
```

The FontMetrics are very important for determining where to place text. The following listing is a method called getTextOrigin used to place text within a rectangle. That is, it accepts a String of text, a Graphics object which will be used to render the text, and a Rectangle to hold the text. The *nFlags* parameter is a bitwise ORing of the flags H_CENTER and V_CENTER. The method returns the x and y coordinates to use as the drawString method's *x* and *y* parameters to place the String within the Rectangle either horizontally or vertically centered (or both).

```
public static final int H_CENTER = 0x00000001;
public static final int V_CENTER = 0x00000002;

public static Point getTextOrigin(String text, Graphics g,
        Rectangle r, int nFlags) {
    FontMetrics fm = g.getFontMetrics();
    int nTypesize = fm.getAscending() + fm.getDescending();
    int nTextwidth = fm.stringWidth(text);

    Point ptRet = new Point(
            nFlags & V_CENTER ? r.width/2 - nTextwidth/2 : 0,
            nFlags & H_CENTER ? r.height/2 - nTypesize/2 :
                    fm.getAscending() );

    return ptRet;
}
```

Figure 7-5 shows a simple Applet which demonstrates the use of getTextOrigin. In a two-by-two grid, the same text is displayed four times. The top left grid cell contains the text without any centering flags, so the text is left and top flushed. The bottom left cell uses only the H_CENTER flag, so the text is flush left but centered top-to-bottom. The text in the top right cell is flush with the top of the cell but centered left-to-right because only the V_CENTER flag is used. The bottom right cell uses both the H_CENTER and V_CENTER flags and the text is centered within the cell.

**Figure 7-5** A getTextOrigin method to center rendered text within a rectangle horizonally or vertically

## Geometric Helper Classes

A Rectangle is represented within the Java API by a size, measured in width and height, and a point of origin. The origin is usually the upper-left corner of the Rectangle, though if you allow for negative widths and heights the origin can be any one of the four corners. The Rectangle class internally stores four variables, which are exposed as public to make use of the Rectangle class easier: *x*, *y*, *width* and *height*. The *x* and *y* members describe the origin of the Rectangle, and the *width* and *height* parameters describe the size of the Rectangle. Positive widths extend to the right of the origin, and positive heights extend downwards from the origin.

The two accompanying classes to Rectangle are Point and Dimension. A Point is made up of an X and a Y distance, and represents a two-dimensional point (simple enough). A Dimension is a two-dimensional vector, and is represented by a *width* and a *height* public member variable.

Dimension objects are used almost exclusively by methods of the Component class to describe the size of a Component object on the screen. Component.size returns a Dimension indicating the width and height of the Component. You pass a Dimension object to a Component's resize method to change the width or height of the Component, as shown here:

```
// Make a Component large by 10 pixels in width and height
Dimension d = myComponent.size();
d.width += 10;
d.height += 10;
myComponent.resize(d);
```

An alternative, and easier, way to do the same thing is

```
Rectangle r = myComponent.bounds();
r.grow(10, 10);
myComponent.reshape(r.x, r.y, r.width, r.height);
```

The Rectangle class includes a rich set of methods to modify a Rectangle's point of origin, width, or height. The Rectangle class' move and translate methods modify a Rectangle by changing its point of origin without modifying its width or height. The move method simply changes the origin to the new x and y coordinates specified in its parameters. To move a Rectangle a relative distance from its current origin, use translate.The current origin of a Rectangle is always available by directly accessing its x and y public member variables, as follows:

```
// Move a Rectangle to the absolute point (10, 10)
Rectangle r = new Rectangle(initX, initY, initWidth, initHeight);
r.move(10, 10);

// Move a Rectangle 10 points to the right, and 10 points down
// from current position
Rectangle r = new Rectangle(initX, initY, initWidth, initHeight);
r.translate(10, 10);
```

The size of a rectangle is modified by Rectangle.resize() that changes the Rectangle's width and height to explicit new values. This is analogous to how Rectangle.move() changes the point of origin to an explicit new location.

Analogous to how translate changes the point of origin by a relative amount, grow makes the Rectangle wider and higher than its current size. The grow method is implemented to keep the Rectangle's center point exactly the same after the operation. The practical effect of growing a Rectangle by dX points in width and dY points in height is to move the origin dX points left and dY points up, and to add 2dX to the Rectangle's width and 2dY to the Rectangle's height. Note that positive dX and dY values passed to grow will actually shrink a Rectangle which has a negative *width* and *height*.

Rectangle.reshape takes both new coordinates for the Rectangle's origin, and new width and height values. That is, you can change any aspect of a Rectangle's placement or dimensions using reshape.

```
// Mirror the Rectangle about the x=y diagonal line
// by swapping x with y, and width with height...
r.reshape(r.y, r.x, r.height, r.width);
```

The Rectangle class provides methods for performing two-dimensional unions and intersections of Rectangles. A union of Rectangles, performed by add, modifies the Rectangle object's origin and dimensions to encompass the smallest rectangular area that contains two different Rectangles. Actually, add does not perform a union in the strict mathematical sense. Figure 7-6 illustrates the difference between Java's add method and a mathematical union operation.

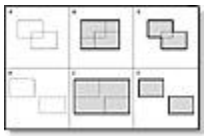

**Figure 7-6** The operation performed by Rectangle.add compared to the result of a mathematical union

An add operation performed on two overlapping rectangles (A) produces a new rectangle (B) precisely sized to encompass both originals, while a mathematical union of the same two rectangles results in a new shape (C) that is not a rectangle at all. Similary, an add operation performed on two non-overlapping rectangles (D), produces another rectangle (E) just large enough to contain the first two, while a union of these rectangles is simply the two disjointed shapes taken together (F).

You can also add a Point to a Rectangle. This has the effect of modifying the Rectangle's origin and dimension to be the smallest rectangular area which encompasses both the origin Rectangle and the Point. Two different overloaded versions of add can be used to

add a Point to a Rectangle: one takes a Point object as its only parameter, and the other takes the X and Y distances as two parameters.

The intersection operation, performed by intersection, modifies a Rectangle's origin and dimensions to encompass the rectangular area of overlap of two different Rectangles. Just as add does not perform a two-dimensional union in the strict sense, the Java intersection operation is not an intersection in the traditional mathematical sense. If there is no area of overlap between the two Rectangles, the resulting Rectangle values have an odd relationship to the original two Rectangles. The resulting rectangular area will have a negative width and height. The best way to describe its position is that it encompasses the area between the two non-overlapping rectangles.

Figure 7-7 shows the difference between Java's intersection operation and a mathematical intersection. For two overlapping Rectangles (A) there is no difference between the Rectangle resulting from Java's intersection operation (B) and the mathematical intersection (C). But for two non-overlapping Rectangles (D), Java's intersection operation results in a non-empty Rectangle (E), while a mathematical intersection would, of course, be the empty set (F).

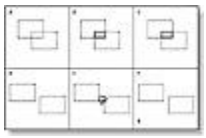

**Figure 7-7** The operation performed by Rectangle.intersection

It is also possible to intersect a Rectangle and a Point, although the process, shown in Figure 7-8, is a bit convoluted. To begin, you essentially negate the original Rectangle (A) by changing the sign of its width and height, and move its origin to the oppposite corner (B). This has the effect of maintaining the same rectangular area as the original Rectangle. Next, you use one of the overloaded versions of add to make a union of the Point with the modified Rectangle (C). Finally change the signs of the resulting rectangle's width and height and again move the origin to the opposite corner of the rectanglur area (D). Strange? No doubt. Here is an implementation of intersection(Point) as described in this paragraph.

```
public class RectangleEx extends Rectangle {
    // Appropriate constructors omitted, but you would want
    // to recreate the large set of constructors available to
    // the Rectangle class.

    public void intersection(Point pt) {
        // Modify this rect by changing the sign of the width
        // and height, and moving the origin point to the
        // opposite corner: keeps same area as original.
        x += width;
        y += height;
        width = -width;
        height = -height;

        // now add this rect with the point.
```

```
            add(pt);

            // Change the signs of the dimensions back, and move
            // the origin to the diagonal corner again.
            x += width;
            y += height;
            width = -width;
            height = -height;
        }
}
```

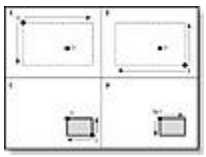

**Figure 7-8** Using the Java API to intersect a Rectangle and a Point

## Graphical Object API Summaries

Table 7-1 lists the Java graphical object classes summarized in this section. Table 7-2 lists the methods for each of these classes, and a short description of each of them.

**Table 7-1** The graphical helper classes

| Class | Description |
|---|---|
| Color | Represents a color in the RGB color model. Methods are provided to easily translate colors between the RGB and HSB (brightness, saturation, hue) color model. |
| Font | Represents a font for rendering text on a display surface. |
| FontMetrics | Stores metrics, describing how a particular Font is rendered on a particular drawing surface. For example, the Font's ascending or descending distances from the baseline, or the width of characters rendered with that Font. |
| Image | An Image is generated from a graphical format file, or from data you provide through an ImageProducer. The Image class implements methods that allow you to draw on Images, just as you would any other display surface (with some restrictions). Informational methods are also provided to supply vital metrics about the image. |
| Point | Encapsulates a two-dimensional coordinate as an X and Y distance. Helper methods for changing the X and Y distances to either an absolute distance or a relative distance. |

| | |
|---|---|
| Rectangle | A Rectangle object describes a rectangle on the two-dimensional plane. A Rectangle is described by a point of origin, a width and a height. Several utility methods are provided with the Rectangle class implementation because the Component class—the basis for all Abstract Windows Toolkit (AWT) windowing classes—uses Rectangles quite a bit. There are several overloaded Rectangle constructors, as well as several methods for manipulating, comparing and combining rectangles via common geometric operations. |
| Polygon | A polygon is represented in Java by a Polygon class instance. A Polygon object stores an ordered set of vertices, describing a multipoint polygon in two dimensions. A couple of utility methods are provided with the Polygon class, although some methods you would expect to be included in the Java API were not. This chapter presents implementations for some of the missing functionality. |

**Table 7-2** Summary of the Color, Font, Images, and Shapes classes and methods

| Class | Method | Description |
|---|---|---|
| Color | getRed, getGreen, getBlue | Gets the value of one of the principal color components. |
| | getRGB | Gets the packed RGB representation of the color. |
| | brighter | Gets a new Color object representing a color brighter than the original. |
| | darker | Gets a new Color object representing a color darker than the original. |
| | getColor | Creates a Color from a hexadecimal String representation of a packed RGB number. |
| | getHSBColor | Creates a new Color object from HSB color component values. |
| | RGBtoHSB | Converts a set of RGB color components to their equivalent HSB color components. |
| | HSBtoRGB | Converts a set of HSB color components to their equivalent RGB color components. |
| Font | getFamily | Gets the text String describing the Font’s font family. |

| | | |
|---|---|---|
| | getName | Gets the text String describing the Font's typeface. |
| | getStyle | Returns a bitfield of flags indicating the additional typeface styles for the Font, such as bold or italics. |
| | getSize | Gets the size of the Font, measured in points. |
| | isPlain | Tells whether any typeface style flags are used by the Font. |
| | isBold | Tells whether the Font is a boldface font. |
| | isItalic | Tells whether the Font is an italics font. |
| | getFont | Static method that creates a Font from just a typeface name. |
| FontMetrics | getFont | Gets the Font that this FontMetrics measures. |
| | getLeading | Gets the suggested leading for the Font. The leading is the suggested spacing between successive lines of text, measured from the top of the tall characters which extend above the baseline (the "ascent") to the bottom of the characters which extend below the baseline (the "descent"). |
| | getAscent | Gets the height of characters above the baseline. |
| | getDescent | Gets the distance below the baseline for characters which hang below the baseline, such as "p", "q", and "j". |
| | getHeight | Gets the suggested distance between successive lines of text, measured baseline-to-baseline. |
| | getMaxAscent | Gets the maximum extension of the tallest character above the baseline. |
| | getMaxDescent | Gets the maximum descension of any character below the baseline. |
| | charWidth | Gets the width in logical units (e.g., in pixels for most display surfaces) for a particular character. |
| | stringWidth | Gets the width in logical units of a string of characters. |
| | charsWidth | Gets the width in logical units of a set of characters presented as an array of |

| | | |
|---|---|---|
| | | chars. |
| | bytesWidth | Gets the width in logical units of a set of characters presented as an array of bytes. |
| | getWidths | Retrieves an array of widths for each character in the ASCII character set. |
| Image | getWidth | Gets the width in pixels of the Image. |
| | getHeight | Gets the height in pixels of the Image. |
| | getSource | Creates an ImageProducer which will deliver the pixel data and ColorModel of this Image to an ImageConsumer or ImageFilter. |
| | getGraphics | Gets a Graphics object using this Image as its drawing surface. Only in-memory Images can successfully use this method. |
| | getProperty | Each Image has an extensible set of properties telling particulars about the format, source, and filtering of the Image. Each property has a unique String name, and the property's value is returned as a human-readable String. |
| | flush | Forces all pixel values for the Image to be forgotten. The next time Image pixel values are accessed, the Java system will reconstruct the Image from its source. |
| Point | move | Changes the X and/or Y position of the point. |
| | translate | Moves the X and/or Y position of the point a specified distance along a particular axis. |
| Polygon | addPoint | Polygon instances are populated with vertices using this method. Note that once a vertex is added to a polygon it can not be removed without directly manipulating the *xpoints* and *ypoints* Polygon member arrays. |
| | getBoundingBox | The smallest rectangle which can contain all the vertices of the Polygon is returned. Note that the Rectangle object returned is actually a member variable of the Polygon object. Direct manipulation of this Rectangle will |

| | | |
|---|---|---|
| | | corrupt it until a new vertex is added to the Polygon. |
| | inside | Tells whether or not a point lies within the Polygon. Uses the even-odd insideness rule to calculate whether or not the point is within the polygon. |
| Rectangle | reshape | In a single method call, this method allows you to change the origin, width, and height of a Rectangle. |
| | resize | Changes the width and height of the Rectangle without modifying the origin point. |
| | move | Modifies the origin point without changing the width and height. |
| | translate | Moves the origin point a specified distance in the X and Y directions without modifying the width nor height. |
| | inside | Tests whether or not a point lies within the Rectangle. |
| | intersects | Tests whether or not another Rectangle intersects this one. |
| | intersection | Computes the rectangle which is an intersection of this one and another Rectangle object. |
| | union | Computes the smallest Rectangle which contains both this Rectangle and another. |
| | add | Adding a rectangle to another Rectangle is the same as a union. Adding a Point to the Rectangle computes the smallest rectangle which contains both this Rectangle and an external Point. |
| | grow | Grows the Rectangle a specific distance in all four directions, such that the center point of the resultant rectangle is the same as the center point of the original. |
| | isEmpty | Tests whether or not the Rectangle has a non-zero volume. That is, whether or not the width and height are both non-zero. |

## Color

**Purpose**

Represents a color as a red, green, and blue color component value.

**Syntax**

public class Color

**Description**

Color objects store the red, green, and blue color components for a single Color. The Graphics class uses Color objects to specify coloring of the foreground, background, and alternate-color (when the Graphics is in XOR mode). Figure 7-9 shows the class hierarchy of the Color class.

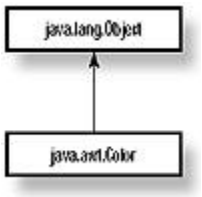


**Figure 7-9** The class hierarchy of the Color class

**PackageName**

*java.awt*

**Imports**

*java.io.*, java.lang.**

**Constructors**

public Color(int red, int green, int blue);
public Color(int rgb);
public Color(float flRed, float flGreen, float flBlue);
Specify the red, green, and blue color components separately as integers or as floats. Alternatively, specify the three color components in a packed int (0xFFrrggbb).

**Example**

Several public member variables of the Color class are static Color objects describing common colors. These members are used as a "shorthand" for specifying common Colors. For example, the two following lines of code are equivalent ways to change a Graphics object's foreground color to yellow.

```
Graphics g;

g.setColor(new Color(255, 255, 0)); // create new Color object.
g.setColor( Color.yellow ); // refer to static yellow Color.
```

The following table lists the 13 static Colors and their RGB values, expressed in hexadecimal values.

| Color | RGB values |
|---|---|
| white | *r*: 0xFF; *g*: 0xFF; *b*: 0xFF |
| lightGray | *r*: 0xC0; *g*: 0xC0; *b*:0xC0 |
| gray | *r*: 0x80; *g*:0x80; *b*: 0x80 |
| darkGray | *r*: 0x40; *g*: 0x40; *b*: 0x40 |
| black | *r*: 0x00; *g*: 0x00; *b*: 0x00 |
| red | *r*: 0xFF; *g*: 0x00; *b*: 0x00 |
| pink | *r*: 0xFF; *g*: 0xAF; *b*: 0xAF |
| orange | *r*: 0xFF; *g*: 0xC8; *b*: 0xC8 |
| yellow | *r*: 0xFF; *g*: 0xFF; *b*: 0x00 |
| green | *r*: 0x00; *g*: 0xFF; *b*: 0x00 |
| magenta | *r*: 0xFF; *g*: 0x00; *b*: 0xFF |
| cyan | *r*: 0x00; *g*: 0xFF; *b*: 0xFF |
| blue | *r*: 0x00; *g*: 0x00; *b*: 0xFF |

## getRed

**ClassName**
Color
**Purpose**
Gets the value of the red color component of this Color.
**Syntax**
public int getRed();
**Parameters**
None.
**Imports**
None.
**Description**
Retrieves the value of the red color component of the Color object. This is the value originally passed as the red color component to the Color class contructor.
**Returns**
The value of the red color component in the range 0-255 is returned.
**See Also**
The getGreen, getBlue, and getRGB methods of the Color class
**Example**

This code sample creates a new Color object that is 75 percent as bright as the original. Each of the red, green, and blue color components are scaled by the value 0.75 (float) and converted to a float value 0-1, which is used by the normalized magnitude version of the Color constructor to create a new, dimmer Color object.

```
Color colorOriginal;

// colorOriginal is initialized to some color.

Color colorDimmer = new Color(
        ((float)colorOriginal.getRed() * 0.75) / 255,
        ((float)colorOriginal.getGreen() * 0.75) / 255,
        ((float)colorOriginal.getBlue() * 0.75) / 255);
```

## getGreen

**ClassName**

Color

**Purpose**

Gets the value of the green color component of the Color.

**Syntax**

public int getGreen();

**Parameters**

None.

**Imports**

None.

**Description**

Retrieves the value of the green color component of the Color object. This is the value originally passed as the green color component to the Color class constructor.

**Returns**

The value of the green color component in the range 0-255 is returned.

**See Also**

The getRed, getBlue, and getRGB methods of the Color class

**Example**

See the code sample for getRed, which also demonstrates use of getGreen and getBlue.

## getBlue

**ClassName**

Color

**Purpose**

Gets the value of the blue color component of the Color.

**Syntax**

public int getBlue();

**Parameters**

None.

**Imports**

None.

**Description**

Retrieves the value of the blue color component of the Color object. This is the value originally passed as the blue color component to the Color class constructor.

**Returns**

The value of the blue color component in the range 0-255 is returned.

**See Also**

The getRed, getGreen, and getRGB methods of the Color class

**Example**

See the sample for getRed, which also demonstrates use of getBlue and getGreen.

## getRGB

**ClassName**

Color

**Purpose**

Gets a packed integer which contains the values of the red, green, and blue color component for this Color.

**Syntax**

public int getRGB();

**Parameters**

None.

**Imports**

None.

**Description**

Retrieves a 32-bit integer of packed bitfields describing the values of the red, green and blue color components of the Color object.

**Returns**

A 32-bit integer of packed bitfields describing the red, green, and blue color components is returned. The bitfields are described by this hexadecimal mask: 0x00rrggbb. That is, the red component mask is 0x00FF0000, the green mask is 0x0000FF00, and the blue mask is 0x000000FF.

**Example**

This method creates a new Color object from an original object, where the output Color is the same as the original Color with the red and blue color components switched. It would probably be even easier to implement this method using getRed and getBlue, but this code sample is sufficient for demonstrating the getRGB method.

```
public Color switchRAndG(Color colorIn) {
    int rgbOut = ((colorIn.getRGB() & 0x00FF0000) >> 16) |
            ((colorIn.getRGB() & 0x000000FF) << 16) |
            (colorIn.getRGB() & 0xFF00FF00);

    return new Color( rgbOut );
}
```

## brighter

**ClassName**

Color

**Purpose**

Creates a new Color object representing a color which is a brighter version of this Color.

**Syntax**

public Color brighter();

**Parameters**

None.

**Imports**

None.

**Description**

Creates a new Color object which is roughly one-and-a-half times as bright as this Color object. Within the Java API, this method is used to highlight beveled edges, such as the edge around a 3D rectangle.

**Returns**

A new Color object representing a color roughly one-and-a-half times as bright as this Color object. That is, each color component of this Color is multiplied by about 1.5 and used to create a new color object. Of course, the maximum of any color component in the new Color object is 255.

**Example**

The method demonstrated here uses arc segments to create a shaded oval on a Graphics object's drawing surface. The shaded oval is drawn with a beveled edge to look "raised" or "lowered" on the drawing surface, similar to draw3DRect. Brighter and darker are used to create colors, implying shaded versions of the Graphics object's foreground color. The draw3DOval method uses the *nThickness* parameter to indicate the thickness of the 3D oval's border.

```
public draw3dOval(Graphics g, Rectangle rectOval,
        boolean fRasied, int nThickness) {
    Color colorBase = g.getColor();
    Color colorNW =
            fRaised ? colorBase.brighter() : colorBase.darker();
    Color colorSE =
            fRaised ? colorBase.darker() : colorBase.brighter();

    // The magnitude of nThickness tells how many concentric
    // 3D ovals to draw.
    for( int i=-nThicknewss/2 ; i<nThickness/2 ; I++ ) {
        Rectangle r = new Rectangle(rectOval.x, rectOval.y,
                rectOval.width, rectOval.height);
        r.grow(i,i);

        // Draw NW sector in colorNW, SE sector in colorSE,
        // and NE, SW sectors in colorBase.
        g.setColor( colorNW );
        g.drawArc(r.x, r.y, r.width, r.height,
                Math.PI, Math.PI/2);
```

```
        g.setColor( colorBase );
        g.drawArc(r.x, r.y, r.width, r.height,
                3*Math.PI/2, Math.PI);
        g.drawArc(r.x, r.y, r.width, r.height,
                Math.,PI/2, 0);

        g.setColor( colorSE );
        g.drawArc(r,x, r.y, r.width. r.height,
                0, 3*Math.PI/2);
    }
}
```

## darker

**ClassName**

Color

**Purpose**

Creates a new Color object representing a color which is a darker version of this Color.

**Syntax**

public Color darker();

**Parameters**

None.

**Imports**

None.

**Description**

Creates a new Color object which is roughly 70 percent as bright as this Color object. Within the Java API, this method is used to highlight beveled edges, such as the edge around a 3D rectangle.

**Returns**

A new Color object representing a color roughly 70 percent as bright as this Color object. That is, each color component of this Color is multiplied by about 0.7 and used to create a new color object.

**Example**

See the code sample for *brighter*.

## getColor

**ClassName**

Color

**Purpose**

Creates a Color from a String representation of packed RGB information.

**Syntax**

public static Color getColor(String nm);
public static Color getColor(String nm, Color v);
public static Color getColor(String nm, int rgb);

**Parameters**

***String nm***

The text value of this String is the decimal or hexadecimal value of a 32-bit integer. This integer has 3 packed 8-bit bitfields representing each of the red, green, and blue color components. The bitfield format is 0x00rrggbb. For example, "00FFFFFF" represents the color white.

***Color v***

The default color to return if the *nm* parameter is incorrectly formatted. That is, if the *nm* parameter does not contain a valid number.

***int rgb***

An integer of 3 packed 8-bit bitfields representing the color components of the default Color to return if the *nm* parameter is incorrectly formatted. That is, if the *nm* parameter does not contain a valid number.

**Imports**

None.

**Description**

Allows you to create a Color object from a text String. The String is a text version of a 32-bit packed RGB value. Overloaded versions exist so you can specify an alternative color if the text String is ill-formated.

**Returns**

A new Color object representing the color components specified by the *nm* String parameter. If the first overloaded version of this method is used, and the *nm* parameter is incorrectly formatted, then null will be returned.

**Example**

The main method of this object prompts the user for a packed RGB integer value for a new Color object. The getColor method is used to build the Color object, and Color.toString is used to display what Color was actually created. "Null" will be displayed if the input text from the user does not represent a valid packed RGB value. Either decimal or hexadecimal notation may be used.

```
import java.awt.Color;

public class TestColorProgram {

    public static void main(String[ ] astrArgs) {
        String strInput = new String();
        String strTerminator = "QUIT";
        String strPrompt = "Color value (\"QUIT\" to end program): ";

        System.out.print( strPrompt );
        strInput = System.in.readln();
        while(!strTerminator.equals(strInput)) {
            Color c = Color.getColor(strInput);

            System.out.println( "Color created is: " +
                     c.toString() + "\n\n" );
            System.out.print( strPrompt );
            strInput = System.in.readln();
        }
    }
}
```

## getHSBColor

**ClassName**

Color

**Purpose**

Creates a new Color object from HSB color components.

**Syntax**

public static Color getHSBColor(float h, float s, float b)

**Parameters**

***float h***

HSB hue color component for the new Color to create. This is measured in radians.

***float s***

0-1 value of the HSB saturation color component for the new Color to create.

***float b***

0-1 value of the HSB brightness color component for the new Color to create.

**Imports**

None.

**Description**

Creates a new Color object from HSB color components. "HSB" is a theoretical color model in which colors are measured by Hues, Saturation and Brightness. (Please refer to a textbook on color theory for a complete discussion of the HSB color model.) Instead of referring to colors in terms of their red, green, and blue color components, as most the other Color class methods do, this method uses hue, saturation, and brightness color components.

**Returns**

A new Color instance is returned, which represents a color with the specified hue, saturation, and brightness color components.

**Example**

This example takes an input Color object and uses it to create a new Color object with a change in the Hue color component of the HSB representation of the Color. Hue is an angle, measured in radians. This method flips the hue color component by π radians (180°). Since hue is measured as an angle, the simplest method to invert the angle by π radians is to invert its sign.

```
public Color InvertHue(Color c) {
    float[ ] aflHSB = Color.RGBtoHSB( c.getRed(),
            c.getGreen(), c.getBlue() );

    aflHSB[0] = -aflHSB[0]; // 0-th value is hue.
    return new Color(Color.HSBtoRGB(afl[0],
            aflHSB[1], aflHSB[2]));
}
```

## RGBtoHSB

**ClassName**

Color

**Purpose**

Converts a set of RGB color components to their HSB equivalents.

**Syntax**

public static float[ ] RGBtoHSB(int r, int g, int b, float[ ] hsbvals);

**Parameters**

***int r***

The red color component of the color to convert to the HSB color model. Must be between 0-255.

***int g***

The green color component of the color to convert to the HSB color model. Must be between 0-255.

***int b***

The blue color component of the color to convert to the HSB color model. Must be between 0-255.

***flaot[ ] hsbvals***

An array of at least three elements. The return values of the conversion are returned in this array. If *hsbvals* is null, then an array of three float values is allocated on behalf of the calling code by Color.RGBtoHSB(). Note that there is no error checking by RGBtoHSB() to ensure the array is at least three elements long. An ArrayIndexOutOfBounds exception will be thrown if this array is not at least three elements long.

**Imports**

None.

**Description**

Converts a set of RGB color components to their equivalent HSB color components. Use this method to convert a color between the RGB and HSB color representation schemes.

**Returns**

The same value passed in the *hsbvals* parameter is returned. If null is passed for *hsbvals*, then the return value is a reference to an array allocated by this method on behalf of the calling code.

**See Also**

The HSBtoRGB method of the Color class

**Example**

See the example for the getHSBColor method.

## HSBtoRGB

**ClassName**

Color

**Purpose**

Converts a set of HSB color components to their RGB equivalents.

**Syntax**

public static int HSBtoRGB(float hue, float saturation, float brightness);

**Parameters**

***float hue***

The hue color component of the color to be converted, measured in radians.

***float saturation***

The saturation color component of the color to be converted. This value is in the range 0-1.

***float brightness***

The brightness color component of the color to be converted. This value is in the range 0-1.

**Description**

Converts a set of HSB color components to their equivalent packed 32-bit integer of RGB color component values. Use this method to convert between the HSB and RGB color representation schemes.

**Returns**

A 32-bit integer of packed RGB color components. The red, green, and blue color components describe the same color as the input *hue*, *saturation*, and *brightness* parameters.

**See Also**

The RGBtoHSB method of the Color class

**Example**

See the example for the getHSBColor method.

## Font

**Purpose**

Represents a Font with which to render text on display surfaces.

**Syntax**

public class Font

**Description**

Instances of the Font class describe a typesetting font for displaying text on a display surface. Fonts are described by a name, size and style. A Font on a particular display surface is measured by a FontMetrics, which tells the size of characters when they are displayed on that display surface.

Different Fonts are available for use on different desktops. The AWT Toolkit object provides the names of available Fonts through Toolkit.getFont.

Figure 7-10 shows the class hierarchy of the Font class.

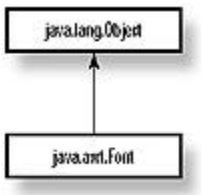


**Figure 7-10** The class hierarchy of the Font class

**Package**

*java.awt*

**Imports**

None.

**Constructors**

public Font(String *name*, int *style*, int *size*);

Specify a font typeface name, style flags ORed together bitwise, and the size of the font in points. The Toolkit class method getFont enumerates all the valid font typeface names.

**Parameters**

The Font class includes several static constants for specifying font styles. Generally, these style flags are ORed together bitwise to specify a series of style indicators for a Font. The following table lists those style constants.

| Constant | Description |
|---|---|
| PLAIN | The absence of other style indicators. |
| BOLD | The boldface style indicator. |
| ITALICS | The italics style indicator. |

**Example**

This example demonstrates how to create Fonts in Java.

```
public void SampleFonts(Graphics g) {
    String astrFontList =
            Toolkit.getDefaultToolkit().getFontList();
    int y = 0;
    Font fontOld = g.getFont();

    for( int i=0 ; i<astrFontList.length ; i++ ) {
        Font f = Font.getFont(astrFont[i]);
        if( null == f ) continue;

        g.setFont(f);
        FontMetrics fm = g.getFontMetrics();

        y += fm.getAscent() + fm.getLeading();

        String strSample = astrFontList[i] + " Aa Bb Cc Dd Ee";
        g.drawText(strSample, 0, y);

        y += fm.getDescent();
}
```

## getFamily

**ClassName**

Font

**Purpose**

Gets the name of the font family that this Font's typeface belongs to.

**Syntax**

public String getFamily();

**Parameters**

None.

**Imports**

None.

**Description**

Returns a String value of the Font's font family. If the font family name is not available, then the Font's typeface name is returned. The font family is stored as a System property under the key "awt.font.<typeface-name>". If this System property is not available, then the Font's typeface name is returned instead.

**Returns**

A String filled with the name of the font family this Font belongs to.

**Example**

This example method prints all available information about a particular Font object, using the various public methods of the Font class.

```
public void print(String s) {
    System.out.println(s);
}

public void printFontInfo(Font f) {
    // Print the Font family
    print("Font family: " + f.getFamily());

    // Print the Font face name.
    print("Font face name: " + f.getName());

    // Print the Font's style attributes
    print("Font is:");
    if(f.isPlain()) {
        print("\tPLAIN");
    } else {
        print("\t" + (f.isBold() ? "BOLD" : "NOT BOLD"));
        print("\t" + (f.isItalic() ? "ITALIC" : NOT ITALIC));
    }
    // Print style flags as demonstrated by the getStyle
    // method.
    print([using getStyle:]");
    if(0 != (f.getStyle() & Font.PLAIN))
        print("\tPLAIN");
    } else {
        print("\t" + (0 != (f.getStyle() & Font.BOLD) ?
                "BOLD" : "NOT BOLD");
        print("\t" + (0 != (f.getStyle() & Font.ITALIC) ?
                "ITALIC" : "NOT ITALIC");
    // Print the font's size
    print("Font size: " + f.getSize());
}
```

## getName

**ClassName**

Font

**Purpose**

Gets the name of the Font's typeface.

**Syntax**

public String getName();

**Parameters**

None.

**Imports**

None.

**Description**

Returns the typeface name for the Font. This String should indicate the same value passed to the Font constructor, or to getFont, whichever was used to create the Font object.

**Returns**

The typeface name of the Font is returned in a String object.

**See Also**

The getFont method of the Font class.

**Example**

See the example for the getFamily method of the Font class.

## getStyle

**ClassName**

Font

**Purpose**

Gets the style flags for the Font.

**Syntax**

public int getStyle();

**Parameters**

None.

**Imports**

None.

**Description**

Gets an integer which describes the style indicators used to create the Font. One or more of the static Font class constants PLAIN, BOLD, or ITALICS will be combined to describe the style of the Font.

**Returns**

The style flags for the Font are ORed together to create the return value for this method. This should be the same value passed to the Font constructor.

**Example**

See the example for the getFamily method of the Font class.

## getSize

**ClassName**

Font

**Description**

Return the size of the Font, measured in points.

**Syntax**

public int getSize();

**Parameters**

None.

**Imports**

None.

**Description**

The size is the combination of the typeface's ascension above the baseline plus its descension below the baseline. The return value is the same as the *size* parameter passed to the Font constructor.

**Returns**

The point size of the Font, measured in "points." A point is a typesetter's unit equal to 1/72 inch.

**Example**

See the example for the getFamily method of the Font class.

## isPlain

**ClassName**

Font

**Purpose**

Tells whether the Font's style is plain or not.

**Syntax**

public boolean isPlain();

**Parameters**

None.

**Imports**

None.

**Description**

Indicates whether the Font's style is devoid of any special flags, such as BOLD or ITALICS.

**Returns**

True is returned if all style flags for the Font are cleared.

**See Also**

The isBold and isItalics methods of the Font class

**Example**

See the example for the getFamily method of the Font class.

## isBold

**ClassName**

Font

**Purpose**

Tells whether the Font is boldface.

**Syntax**

public boolean isBold();

**Parameters**

None.

**Imports**

None.

**Description**

Tells whether or not the BOLD style flag for the Font is set.

**Returns**

Returns true if the *Font.BOLD* style flags is set for the Font.

**See Also**

The isPlain and isItalics methods of the Font class

**Example**

See the example for the getFamily method of the Font class.

## isItalic

**Class**

Font

**Purpose**

Tells whether the Font is italicized.

**Syntax**

public boolean isItalic();

**Parameters**

None.

**Imports**

None.

**Description**

Tells whether or not the ITALICs style flag for the Font is set.

**Returns**

Returns true if the *Font.ITALICs* style flag is set for the Font.

**See Also**

The isPlain and isBold methods of the Font class

**Example**

See the example for the getFamily method of the Font class.

## getFont

**ClassName**

Font

**Purpose**

Creates a new Font given just a typeface name.

**Syntax**

public static Font getFont(String *nm*);

public static Font getFont(String *nm*, Font *font*);

**Parameters**

***String nm***

The name of the typeface to use for the Font. Only valid typeface names are acceptable. A list of valid typeface names for the local system can be accessed through Toolkit.getFontList() which returns an array of typeface names stored in Strings.

***Font font***

Default Font object to return if the typeface name passed in the *nm* parameter is not a valid typeface name.

**Description**

This static method creates a new Font object given just a typeface name. In addition to the typeface name, you can add style attributes to the Font by prepending style prefixes to the typeface name in the *nm* parameter.

**Returns**

A new Font object which uses the typeface indicated by the *nm* parameter is returned. Null will be returned by the first overloaded version of this method if the *nm* parameter does not indicate a valid typeface name.

**Example**

This example program asks the user for the name of a typeface to display. Once provided, a Font of that typeface is created using getFont and is used to display a sample string in a Label centered in a floating Frame.

```
import java.awt.*;

public class GetFontTest {

    public static void main(String[ ] astrArgs) {
        Frame _frame = new Frame( "GetFontTest Program" );
        Label _label = new Label( "This is a test." );

        _frame.setLayout(new BorderLayout());
        _frame.add( "Center", _label );

        _frame.show();

        String strInput = new String();
        String strTerminator = "QUIT";
        String strPrompt = "Font to use (\"QUIT\" to end): ";

        System.out.print( strPrompt );
        strInput = System.in.readln();
        while(!strTerminator.equal(strInput)) {
            Font f = Font.getFont(strInput);
            if( null == f ) {
                System.out.println(strInput +
                        " is not a valid typeface name");

                continue;
            }

            _label.setFont(f);
            _frame.repaint();

            System.out.print( strPrompt );
            strInput = System.in.readln();
        }
```

```
    }
```

## }FontMetrics

**Purpose**

Encapsulates vital metrics of a Font as it is rendered on a specific display surface.

**Syntax**

public abstract class FontMetrics

**Description**

A Font object alone does not contain enough information to compute the size of characters when they are displayed using that Font. To understand why this information may not be available, remember that Font objects only store the Font's point size. But "point size" does not directly translate into a *logical* display size before the Font is actually associated with a particular display surface. For example, imagine trying to use a 30-point Courier Font to display text on a dot matrix printer. The printer probably can only display text using a single text size. In this case, it doesn't matter whether your Font is 30 points or 3 points in size, both will produce visible text of exactly the same size.

Once a particular Font is associated with a display surface (that is, a Graphics object which controls the display surface), then the logical size of the Font's characters on that display surface can be calculated. The FontMetrics class represents the actual size of displayed text on a particular display surface using a particular Font. The various methods of this class automatically compute the width and height of one or more displayed characters for that display surface and Font.

To create a FontMetrics object, you associate the Font you want to use with the Graphics object associated with the desired display surface. For example, to calculate the width of the string "Measure Me!" as it is displayed on a Canvas using a particular Font, you would use code similar to this

```
public class MyCanvas extends Canvas {
    Font _fontDisplay; // Font to use when printing text.
    String _strTest = "Measure Me!";

    public MyCanvas() {
        // Obtain a reference to the Font which should be used
        // to display text. Store this value in _fontDisplay.

        setFont( _fontDisplay );
    }

    public void paint(Graphics g) {
        // _fontDisplay has already been associated with this
        // Graphics. Obtain FontMetrics.
        FontMetrics fm = g.getFontMetrics();

        // Measure the test string.
        int width = fm.stringWidth( strTest );
    }
}
```

Java was originally designed to be a Unicode language. “Unicode,” for those unfamiliar with the term, is a 16-bit character set designed to encode all the character sets of all languages in the world. The 8-bit ASCII character set is actually a subset of the Unicode set. That is, all ASCII codes translate directly to Unicode value by zero-padding the ASCII value with another byte containing 0. Unfortunately, the Java 1.0 API does not include Unicode support beyond the normal ASCII character set. All FontMetrics methods which refer to character values will only accept ASCII code values. In fact, most FontMetrics methods assume the 8-bit ASCII character set only is being used. In later versions of Java, this API will have to be changed to accommodate the full Unicode character set. Figure 7-11 shows the class hierarchy for the FontMetrics class.

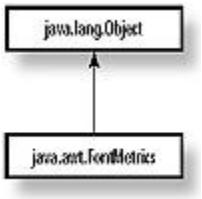


**Figure 7-11** The class hierarchy for the FontMetrics class

**Package**

*awt.java*

**Imports**

None.

**Constructors**

public FontMetrics(Font *font*);

This constructor creates a FontMetrics measuring the Font *font* on the default display device (the desktop). The Graphics method getFontMetrics can be used to get a FontMetrics for the display surface associated with a particular Graphics object.

**Parameters**

None.

**Example**

See the above example, which demonstrates creation and use of a FontMetrics object, to measure the size of a String as it is rendered on the desktop.

## getFont

**ClassName**

FontMetrics

**Purpose**

Gets the Font this FontMetrics was created to measure.

**Syntax**

public Font getFont();

**Parameters**

None.

**Imports**

None.

**Description**

Retrieves the Font object associated with this FontMetrics object. This will be the same Font passed to the FontMetrics constructor, or the same Font selected into the Graphics object which created this FontMetrics through Graphics.getFontMetrics.

**Returns**

The Font object associated with this FontMetrics is returned.

**Example**

This example method ensures that the Graphics object it is passed has a bold font as its current font.

```
public void selectBoldFont(Graphics g) {
    FontMetrics fm = g.getFontMetrics();
    Font f = fm.getFont();
    if(!f.isBold())
        f = new Font(f.getName(), f.getStyle() & Font.BOLD,
                f.getSize());
    g.setFont(f);
}
```

## getLeading

**ClassName**

FontMetrics

**Purpose**

Gets the leading distance between successive lines of rendered text.

**Syntax**

public int getLeading();

**Parameters**

None.

**Imports**

None.

**Description**

Returns the internal leading distance between successive lines of text for the associated Font, as rendered on the associated display surface. The “internal leading” distance is the suggested distance from the bottom of characters descending below the text baseline to the top of characters ascending above the baseline of the subsequent line of text.

**Returns**

The internal leading value for the associated Font rendered on the associated display surface is returned. The internal leading is measured in logical units of the display surface (e.g., in pixels for the on-screen desktop).

**Example**

This example uses the internal leading value as an inset distance to leave around a String of text rendered on a display surface.

```
public class MyStringDisplayComponent extends Canvas {
    String _str;
    public MyStringDisplayComponent(String str) {
```

```
        _str = =str;
    }
    public Dimension preferredSize() {
        FontMetrics fm = getGraphics().getFontMetrics();
        Rectangle r = new Rectangle(0, 0, fm.stringWidth(_str),
                fm.getAscent() + fm.getDescent());
        r.grow(fm.getLeading());
        return r.size();
    }
    public void paint(Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        g.drawString(_str, fm.getLeading(),
                fm.getLeading() + fm.getAscent());
        return;
    }
}
```

## getAscent

**ClassName**

FontMetrics

**Purpose**

Gets the ascending distance of tall characters.

**Syntax**

public int getAscent();

**Parameters**

None.

**Description**

Retrieves the ascent of tall characters above the baseline. The getMaxAscent() method returns the ascent of the highest character, while this method returns the ascent of the majority of tall characters (such as “t”, “h” or “k”).

**Returns**

Returns the ascent of the Font as rendered on the associated display surface. The ascent is measured in logical units of the display surface (e.g., for the desktop, the return value would be in pixels).

**Example**

See the example for the getLeading method.

## getDescent

**ClassName**

FontMetrics

**Purpose**

Gets the descending distance of characters that hang below the baseline.

**Syntax**

public int getDescent();

**Parameters**

None.

**Imports**

None.

**Description**

Retrieves the descent of characters that hang below the baseline. The getMaxDescent() method returns the descent of the character that hangs lowest below the baseline for this Font, while this method returns the descent of the majority of characters which hang below the baseline (such as “q”, “j” or “g”).

**Returns**

Returns the descent of the Font as rendered on the associated display surface. The descent is measured in logical units of the display surface (e.g., for the desktop the return value would be in pixels).

**Example**

See the example for the getLeading method.

## getHeight

**ClassName**

FontMetrics

**Purpose**

Gets the total height of a single line of rendered text.

**Syntax**

public int getHeight();

**Parameters**

None.

**Imports**

None.

**Description**

Retrieves the total height of a line of text drawn using the associated Font on the associated display surface. This value is the sum of the ascent, the descent, and the internal leading. The height of a line is a convenient measure of the distance bewteen baselines of successive lines of text.

**Returns**

The total height of a line of text rendered on the associated display surface using the associated Font. This height is measured in logical units of the display surface (e.g., for the desktop the return value would be in pixels.

**Example**

The paint method of this example Component paints two lines of text on the passed Graphics, using the height as a recommended distance between baselines.

```
String _str1;
String _str2;

// _str1 and _str2 must be initialized somewhere before
// paint() is called.

public void paint(Graphics g) {
    Fontmetrics fm = g.getFontMetrics();
```

```
    int y = fm.getAscent() + getLeading();
    g.drawString(_str1, 0, y);
    g.drawString(_str2, 0, y + fm.getHeight());
}
```

## getMaxAscent

**ClassName**

FontMetrics

**Purpose**

Gets the maximum ascent above the baseline of any chartacter in the character set.

**Syntax**

public int getMaxAscent();

**Parameters**

None.

**Imports**

None.

**Description**

This method retrieves the maximum ascent above the baseline of any character in the character set, rendered on the associated display surface using the associated Font. The getAscent method retrieves the ascent of the majority of "tall" characters such as "t", "h" and "k".

**Returns**

The maximum ascent of all characters and the character set if rendered on the associated display surface using the associated Font. The maximum ascent is measured in logical units of the display surface (e.g., pixels for the on-screen desktop).

**See Also**

The getMaxDescent method of the FontMetrics class

**Example**

This example method calculates the *maximum* height of a Font, based on the Font's maximum ascent and maximum descent. (The normal getHeight method of the Font class calculates the Font's height based on the average ascent and descent of the Font.)

```
public int getMaxHeight(Font f) {
    return f.getLeading() + f.getMaxAscent() +
            f.getMaxDescent();
}
```

## getMaxDescent

**ClassName**

FontMetrics

**Purpose**

Get the maximum descent below the baseline of any character in the character set.

**Syntax**

public int getMaxDescent(); public int getMaxDecent();

**Parameters**

None.

**Imports**

None.

**Description**

This method retrieves the maximum descent below the baseline of any character in the character set, rendered on the associated display surface using the associated Font. The getDescent() method retrieves the descent of the majority of characters such as “q”, “j”, and “g”. Note that another version of this same method exists which is misspelled. The misspelled version maintains backwards compatibility with alpha and beta versions of the Java API, which were developed before spellcheckers.

**Returns**

The maximum descent of all characters and the character set if rendered on the assoicated display surface using the associated Font. The maximum descent is measured in logical units of the display surface (e.g., pixels for the on-screen desktop).

**See Also**

The getMaxAscent method of the FontMetrics class.

**Example**

See the example for the getMaxAscent method of the FontMetrics class.

## charWidth

**ClassName**

FontMetrics

**Purpose**

Gets the width of a specific character.

**Syntax**

public int charWidth(char ch); public int charWidth(int ch);

**Parameters**

***char ch***

Char storing the ASCII code of the character to measure. By design, Java is a Unicode language which uses 16-bit Unicode character values. Currently, however, only the 8-bit ASCII code values are recognized by Java.

***int ch***

32-bit integer storing the ASCII code of the character to measure. When later versions of Java gain fuller Unicode capabilities, the overloaded version of this method, which takes an integer parameter, will have to be used to process 16-bit Unicode values. You can use the overloaded version of this method with ASCII code values, zero-padded in the top three bytes of this integer parameter.

**Imports**

None.

**Description**

Retrieves the width of a single character rendered in the associated Font on the associated display surface.

**Returns**

The width (in logical untis of the associated display surface) of a single character.

**Example**

This example creates an Image object precisely the same size as the character which is also painted on the Image surface.

```
public Image getCharImage(char ch) {
    Graphics g = getGraphics();
    g.setFont(getFont());
    FontMetrics fm = g.getFontMetrics();

    Image img = createImage(fm.charWidth(ch), fm.getAscent() +
            fm.getDescent);
    Graphics img_g = img.getGraphics();
    g.setFont(getFont());
    g.drawString(new String(ch));
    return;
}
```

## stringWidth

**ClassName**

FontMetrics

**Purpose**

Calculates the width of a String of text.

**Syntax**

public int stringWidth(String str);

**Parameters**

***String str***

The String of text to measure.

**Imports**

None.

**Description**

Calculates the width of the String of text rendered on the associated display surface using the associated Font.

**Returns**

The total width of the String of text is returned. The return value is in logical units of the associated display surface (e.g., for the desktop, the return value is in pixels).

**Example**

A useful method is getMeasuredSubstring(), listed below. This method breaks the input String at the last whitespace chartacter which can fit within a particular width on the default display surface. This is useful for finding the correct place to break a string when word-wrapping text yourself (instead of depending on a TextArea to do it).

```
public int getMeasuredSubstring(String str, Font font, int width) {
    FontMetrics fm = new FontMetrics(font);
    String strRet = new String();
    String strToken = new String();
    int widthAccum = 0;

    StringTokenizer st = new StringTokenizer(str);
```

```
    while( widthAccum + fm.stringWidth(strToken) < width ) {
        strRet += strToken();
        widthAccum += fm.stringWidth(strToken);

        if( !st.hasMoreTokens() )
            break;
        strToken = st.nextToken();
    }
    return strRet;
}
```

## charsWidth

**ClassName**

FontMetrics

**Purpose**

Calculates the width of a set of characters stored in an array of characters.

**Syntax**

public int charsWidth(char[ ] data, int off, int len);

**Parameters**

***char[ ] data***

Array of characters to measure. Each element holds an 8-bit ASCII character value.

***int off***

Zero-based index of the first character to measure in the *data* array.

***int len***

Count of characters to measure in the *data* array.

**Imports**

None.

**Description**

Calculates the width of characters in a char array on the associated display surface using the associated Font object to render the characters. You specify the offset into the array and the number of characters to measure. Use 0 for the *offset* and data.length for the *len* to measure the entire array of characters.

Note that if the data array passed into this method is not at least *(off+len)* elements in length, then this method will throw an ArrayIndexOutOfRange exception.

**Returns**

The sum of the widths of the *off* through *(off+len-1)* characters in the *data* array. Character measurements are in logical units of the associated display surface (e.g., in pixels for the on-screen desktop).

**Example**

This method is very similar to the stringWidth method. See the example for stringWidth.

## bytesWidth

**ClassName**

FontMetrics

**Purpose**

Calculates the width of a set of characters stored in an array of bytes.

**Syntax**

public int bytesWidth(byte[ ] data, int off, int len);

**Parameters**

***byte[ ] data***

Array of characters to measure. Each element holds an 8-bit ASCII character value.

***int off***

Zero-based index of the first character to measure in the *data* array.

***int len***

Count of characters to measure in the *data* array.

**Imports**

None.

**Description**

Calculates the width of all the characters in a byte array on the associated display surface, using the associated Font object to render the characters. Since characters are stored in bytes, which are always eight bits wide, only ASCII characters can be measured using this method, even in future version of Java.

Note that if the data array passed into this method is not at least *(off+len)* elements in length, then this method will throw an ArrayIndexOutOfRange exception.

**Returns**

The sum of the widths of the *off* through *(off+len-1)* characters in the *data* array. Character measurements are in logical units of the associated display surface (e.g., in pixels for the on-screen desktop).

**Example**

This method is very similar to the stringWidth method. See the example for the stringWidth method.

## getWidths

**ClassName**

FontMetrics

**Purpose**

Gets the widths of all characters in the ASCII character set.

**Syntax**

public int[ ] getWidths();

**Parameters**

None.

**Imports**

None.

**Description**

Gets an array of the widths of each character in the ASCII character set, as rendered with the associated Font on the associated display surface.

**Returns**

A 256 element array of integer values, each element containing the width of the corresponding character in the ASCII character set. The width of each character is measured in logical units of the display surface. Note that only the width of the ASCII characters is returned. Even though Java is designed to be a Unicode language, only the ASCII character set is supported by the Java 1.0 API.

**Example**

This is an example implementation of the FontMetrics.charsWidth method which utilizes getWidths.

```
public class MyFontMetrics extends FontMetrics {

    public int charsWidth(char[ ] data, int off, int len) {
        int nRet = 0;
        char[ ] achWidths = getWidths();

        for( int i=0 ; i<len ; i++ )
            nRet += achWidths[data[i+off]];

        return nRet;
    }
}
```

## Image

**Purpose**

Represents an image which can be rendered on a Graphics object's display surface.

**Syntax**

public abstract class Image

**Description**

Lots of functionality in the Java API is centered around creating, manipulating, and displaying images. Images are represented in Java as Image class instances. Image objects can represent either a single static picture, or a series of animation frames. The Graphics.drawImage method is built to display both types of Images on a display surface. A couple of chapters in this book deal with other Java classes that handle Image objects. Chapter 1 explains in detail the Graphics class, and touches on the ImageObserver interface which is required to render an Image on a display surface. Figure 7-12 shows the Image class hierarchy.

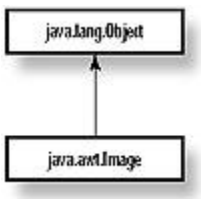


**Figure 7-12** The class hierarchy of the Image class

**Package**

*java.awt*

**Imports**

*java.awt.image.ImageProducer, java.awt.image.ImageObserver*

**Constructors**

None.

No constructor is defined for the Image class, so you cannot create Image objects directly using the *new* operator. Instead, four techniques for creating Image objects indirectly are provided by the AWT API.

First, Image objects can be created by the AWT Toolkit from graphical format files. Image data is generally stored in formatted files, such as GIF or JPEG files. The AWT's Toolkit class exposes methods to create Image objects from popular graphical image formats. Files in one of these formats can be located either on the local file system, or anywhere on the network which can be addressed by a URL. Chapter 3 explains the Toolkit class in detail and explains these methods. For a list of the file formats which can be read by the AWT Toolkit, see the explanation of the Toolkit.createImage() method.

The second method for creating Image objects involves using the ImageProducer interface (discussed in Chapter 8). The ImageProducer API defines an object which can provide image pixel data in a standard way. Image pixel data is provided as individual pixel values and a ColorModel. The ColorModel explains how to convert each pixel's value into RGBA data. The AWT Toolkit class provides an overloaded version of the createImage() method, which creates an Image object from the pixel data provided by an ImageProducer.

The third method for creating an Image object is by using an ImageFilter. The ImageFilter accepts the pixel data from an ImageProducer and modifies that pixel data to create a filtered Image. For example, you might want to convert a particular Image by blurring it or sharpening it. To do this, you would define a BlurImageFilter, which will be used by the AWT Toolkit to convert a source image into a blurred image. The technique for defining and using an ImageFilter is explained in detail in Chapter 8.

Finally, and most simply, the AWT Toolkit also provides an overloaded version of createImage(), which allows you to create a blank Image of arbitrary width and height. Images created this way are called "in-memory" Images. In-memory Images can be drawn on, just like a display surface. This is done by acquiring a Graphics object associated with the Image as its display surface. You then use the Graphics' drawing methods to draw on the Image itself. Only in-memory Images can be drawn on this way. Images created using any one of the other three Image-creation techniques cannot be associated with a Graphics object as a display surface.

Except for in-memory Images, which are just blank, Images must be constructed from a pixel data source. This source may be a graphical formatted file, an ImageProducer you have implemented yourself, or an ImageFilter coupled with an ImageProducer. In any one of these three cases, the Java runtime system actually compiles the Image's pixel data *only as it is needed*. That is, the Image pixel data is only read into memory allocated on behalf of the Image object when

a query is performed which requires the pixel data. For example, an Image created from a graphical file URL will just start to read in its pixel data asynchronously when you ask for the Image to be rendered using Graphics.drawImage.

**Parameters**

None.

**Example**

This example uses the Toolkit's getImage method to read an image from the local file system.

```
public class MyComponent extends Canvas {
    Image _img;
    ...

    public MyComponent(String filename) {
        _img = Toolkit.getDefaultToolkit().getImage(filename);
    }

    public MyComponent(URL urlImage) {
        _img = Toolkit.getDefaultToolkit().getImage(utlImage);
    }

    ...
}
```

## getWidth

**ClassName**

Image

**Purpose**

Gets the width of the Image in pixels.

**Syntax**

public int getWidth(ImageObserver observer);

**Parameters**

***ImageObserver observer***

Object to receive asynchronous notification of the construction of this Image's pixel data if the data has not yet been read into memory.

**Imports**

*java.awt.image.ImageObserver*

**Description**

Retrieves the width of the Image in pixels. Requires an ImageObserver since the Image may have to be constructed asynchronously from its source.

**Returns**

The width of the Image in pixels is returned. If the Image pixel data has not yet been constructed in memory, then the asynchronous construction process is kick-started right away, and -1 is returned by this method. The ImageObserver *observer* will be notified at a later time when the Image's width is available.

**Example**

The ImageCanvas class is a visual Component which merely displays an image on its surface. The constructor to ImageCanvas requires an Image object that is to be displayed. The ImageCanvas' preferredSize method attempts to access the Image's width and height. The ImageCanvas implements the ImageObserver interface in case asynchronous construction of the Image is required.

```
public class ImageCanvas extends Canvas
        implements ImageObserver {
    private Image _img = null;

    // Image object for this canvas to display
    // must be passed in.
    public ImageCanvas(Image img) {
        _img = img;
    }

    // Attempt to get _img Image width and height. If
    // asynchronous construction is required, return a
    // preferred size of (0, 0).
    public Dimension preferredSize() {
        Dimension d = new Dimension(0, 0);
        if( null == _img ) return d;

        int nWidth = _img.getWidth(this);
        int nHeight = _img.getHeight(this);
        if((-1 == nWidth) || (-1 == nHeight)) return d;

        d = new Dimension(nWidth, nHeight);
        return d;
    }

    // Just display the _img Image.
    public void paint(Graphics g) {
        if( null == _img ) return;

        g.drawImage(_img, 0, 0, this);
    }

    // Default implementation of imageUpdate() forces a repaint
    // always. We only want to do an actual repaint if the
    // _img Image has been completely constructed, indicated
    // by the ALLBITS flag of the infoflags parameter.
    public void imageUpdate(Image img, int infoflags,
            int x, int y, int width, int height) {
        if(_img != img) return;

        if( 0 != (infoflags & ALLBITS) )
            super.imageUpdate(img, infoflags, x, y,
                    width, height);
    }
}
```

## getHeight

### ClassName

Image

**Purpose**

Gets the height of the Image in pixels.

**Syntax**

public int getHeight(ImageObserver observer);

**Parameters**

***ImageObserver observer***

Object to receive asynchronous notification of the construction of this Image's pixel data, if the data has not yet been read into memory.

**Imports**

*java.awt.image.ImageObserver*.

**Description**

Retrieves the height of the Image in pixels. Requires an ImageObserver since the Image may have to be constructed asynchronously from its source.

**Returns**

The height of the Image in pixels is returned. If the Image pixel data has not yet been constructed in memory, then the asynchronous construction process is kick-started right away, and -1 is returned by this method. The ImageObserver *observer* will be notified at a later time when the Image's height is available.

**Example**

See the example under the method getWidth of the Image class.

## getSource

**ClassName**

Image

**Purpose**

Gets an ImageProducer that can deliver the pixel data from this Image to an ImageConsumer.

**Syntax**

public abstract ImageProducer getSource();

**Parameters**

None.

**Imports**

*java.awt.image.ImageProducer*

**Description**

Creates a new ImageProducer which will pass this Image's pixel data to any ImageConsumer or ImageFilter. This method is most often used in conjunction with a FilteredImageSource and an ImageFilter to create a filtered version of the Image's pixel data, as demonstrated by the example code below.

**Returns**

An ImageProducer object is returned. This ImageProducer will deliver the pixel data and ColorModel for this Image to any ImageConsumer or ImageFilter it is associated with.

**Example**

The typical use of getSource is to create an ImageProducer to associate with an ImageFilter. This allows you to create a filtered version of the Image. The code below demonstrates exactly how you do this using a FilteredImageSource object.

```
Image imgBase;
ImageFilter filter;

// imgBase is made a reference to a valid Image object.
// filter is made a reference to a valid ImageFilter.

Image imgFiltered = new FilteredImageSource(imgBase.getSource(),
filter);
```

## getGraphics

**ClassName**

Image

**Purpose**

Gets a Graphics object which uses this Image as its display surface.

**Syntax**

public Graphics getGraphics();

**Parameters**

None.

**Description**

Retrieves a Graphics object associated with this Image. This Image object is the display surface for the Graphics, so that all rendering operations performed with the Graphics will be drawn on this Image. Only in-memory Images, created with the AWT Toolkit's overloaded createImage*(width, height)* method, will successfully retrieve a Graphics object. If you try to get the Graphics for an Image created using another technique, then this method will throw an exception. Image.getGraphics is closely associated with the double-buffered rendering technique. The double-buffered rendering technique, for fast visual updating, utilizes an in-memory Image. All drawing operations are performed on the in-memory Image, through the Graphics retrieved by this method. When all graphics have been completely rendered, the Image is copied in full to the on-screen desktop display surface.

**Returns**

A Graphics object which uses this in-memory Image as its display surface.

**Example**

This example demonstrates double-buffering. In this example, the DoubleBufferCanvas' paint method performs several drawing operations on an in-memory Image. When all drawing operations on the in-memory Image are complete, the entire Image is copied to the on-screen display surface. Note that the DoubleBufferCanvas, which is a Component, acts as an ImageObserver for the drawImage operation.

```
public class DoubleBufferCanvas extends Canvas {
```

```
    public void paint(Graphics g) {
        // Get size of this canvas, which is the size of the
        // in-memory Image to create.
        Dimension d = size();
        Image imgInMemory = createImage(d.width, d.height);

        // Perform several drawing operations on the in-memory
        // Image using a Graphics associated with it.
        Graphics gInMemory = imgInMemory.getGraphics();
        gInMemory.fillRect(0, 0, d.width, d.height);
        gInMemory.setXORMode( Color.black );
        gInMemory.fillRect(5, 5, d.width-10, d.height.10);
        gInMemory.fillRect(10, 10, d.width-20, d.height-20);
        gInMemory.fillOval(10, 10, d.width-20, d.height-20);
        // A bunch more operations...

        // Now copy the in-memory Image, which has all drawing
        // operations rendered on it, to the on-screen desktop.
        g.drawImage(imgInMemory, 0, 0, this);
    }

}
```

## getProperty

**ClassName**

Image

**Purpose**

Gets one of the properties stored with the Image.

**Syntax**

public Object getProperty(String name, ImageObserver observer);

**Parameters**

***String name***

Name of the property to retrieve. Individual properties are specific to the image format. Only three properties are publicly defined for the Java API, as indicated in the table below.

***ImageObserver observer***

If no properties have been read in for this Image yet, the ImageObserver will be notified about the property value asynchronously.

**Description**

An extensible list of Image properties is stored within the Image object. Each property has a text name, which is passed into this method to identify the property to retrieve. A property's value is an arbitrary Object.

A comprehensive list of the properties for different image formats is not available as of this book's publication. Also, there is no simple way to poll the Image for a list of its valid properties. The table below lists the three Image properties currently documented in the Java API.

| **Property** | **Description** |
|---|---|
| "comments" | This property's value is a String which can describe the |

| | |
|---|---|
| | Image, hold a copyright notice, or relay any textual information about the Image. |
| “filters” | A concatenation of human-readable Strings (into a single String) listing the ImageFilters used in creating the Image. |
| “croprect” | If a CropImageFilter is used to create the Image, then the “croprect” property holds the boundaries of the rectangle of the original Image that was cropped. |

---

**Returns**

The value of the named property. Note that if the property does not exist for the Image, then an Object class instance is returned (an Object object). If the properties for the Image have not yet been retrieved from the Image’s source, then null will be returned and the object referenced by the *observer* parameter will be notified asynchronously when the Image’s properties have been retrieved.

**Example**

This example prints an Image’s values for the three known Image properties.

```
public void printImageProps(Image img, ImageObserver io) {
    if(null != img.getProperty(“comments”, io))
        System.out.println(“comments: ” +
                img.getProperty(“comments”, io));

    if(null != img.getProperty(“filters”, io))
        System.out.println(“filters: ” +
                img.getProperty(“filters”, io));

    if(null != img.getProperty(“croprect”, io))
        System.out.println(“croprect: ” +
                img.getProperty(“croprect”, io));
}
```

## flush

**ClassName**

Image

**Purpose**

Resets the constructed version of the Image.

**Syntax**

public void flush();

**Parameters**

None.

**Description**

All pixel data for the Image, which has been constructed, is flushed from the system. After a call to flush(), the Image is as if it had just been created and none of the image pixel data had been constructed yet. Any query of the Image’s value would cause the Image to be reconstructed from scratch.

**Returns**

None.

**Example**

This example constantly reloads and redisplays an Image on a Canvas-derived component. This could be useful for doing Web server-based animation. (This particular example would not work for non-interlaced images, since interlaced images cause multiple paint calls to be made while the image is rendered in memory.)

```
public class ServerBasedAnimator extends Canvas {
private Image _img;

    public ServerBasedAnimator(URL url) {
        _img = getImage(url, this);
    }

    public void paint(Graphics g) {
        g.drawImage(_img);
        _img.flush();
        repaint();
    }
}
```

## Point

**Purpose**

The Point class represents a two-dimensional point.

**Syntax**

public class Point

**Description**

A coordinate on the two-dimensional integer plain is abstracted by the Point class. The Point is comprised of an x and y coordinate value. Several public methods are also defined in the Point class for modifying these coordinate values.

**Package**

*java.awt*

**Imports**

None.

**Constructors**

public Point(int x, int y);

The Point object's *x* and *y* members are assigned the values of the x and y parameters to the constructor.

**Parameters**

The following table lists the Point class' public member variables.

| Member | Description |
|---|---|
| int x | The distance along the X axis from the origin to this Point. |
| int y | The distance along the Y axis from the origin to this Point. |

## move

**ClassName**
Point
**Purpose**
Sets the x and y values of the Point object.
**Syntax**
public void move(int x, int y);
**Parameters**
***int x***
New X value for the Point.
***int y***
New Y value for the Point.
**Description**
Changes the *x* and *y* values for the Point object.
**Example**
See the example for the method translate.

## translate

**ClassName**
Point
**Purpose**
Adds a *dx* and a *dy* value to the Point's *x* and *y* member variables.
**Syntax**
public void translate(int dx, int dy);
**Description**
Moves the Point a specified distance along either axis.
**Parameters**
***int dx***
Relative distance along the X axis to move the Point.
***int dy***
Relative distance along the Y axis to move the Point.
**Example**
The following example method moves the Point about the origin in a circle at the distance specified by the *R* parameter. Both move and translate are utilized.

```
public void CircularMotion(Point p, int R) {
    // Start the point at a 3pi/4-radian angle (270 degrees),
    // a distance R from the origin.
    p.move(0, R);

    for( float flRads=0.0 ; flRads<(2*Math.PI)
            ; flRads+=Math.PI/100 ) {
        p.translate((int)(Math.cos(flRads)*R/100),
                (int)(Math.sin(flRads)*R/100));
}
```

## Polygon

**Description**

A Polygon is stored in Java as an ordered set of points on a two-dimensional plane. Each point represents a vertex of the polygon. The Polygon class includes some convenience methods to make polygons easier to work with. Also, the Graphics class includes two methods, fillPolygon and drawPolygon, so that you can draw your Polygon objects on display surfaces.

Consistent with the other geometric helper classes (Point and Rectangle), the Polygon class exposes its member variables with no protection.

Figure 7-13 shows the class hierarchy for the Polygon class.

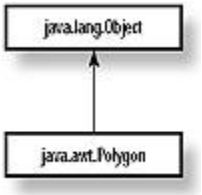


**Figure 7-13** Class hierarchy for the Polygon class

**Package**

*java.awt*

**Imports**

None.

**Constructors**

public Polygon();

public Polygon(int xpoints[ ], int ypoints[ ], int npoints);

The parameterless constructor creates a Polygon with no vertices. You must use the Polygon method addPoint to populate the Polygon with vertices. The second constructor takes an initial list of vertices, specified by an array of x coordinates and an array of y coordinates. When using this constructor, make sure your *xpoints* and *ypoints* arrays have at least *npoints* elements in them, or else an ArrayIndexOutOfBounds exception will be thrown.

**Parameters**

The following table lists the public member variables of the Polygon class.

| Member | Description |
|---|---|
| int[ ] xpoints | An array of x coordinates. |
| int[ ] ypoints | An array of y coordinates. The *n*th element of the *xpoints* array and the *ypoints* array, taken together, indicates the location of the *n*th vertex of the Polygon. Both the *xpoints* array and the *ypoints* array are dynamically reallocated to make more room for new vertices as necessary. |
| int npoints | The number of vertices in the Polygon. The lengths of the *xpoints* and *ypoints* arrays are guaranteed to be at least *npoints* each. |

**Example**

This example creates a Polygon whose vertices are in a star pattern.

```
public Polygon makeStar(int radius) {
    Point pt = new Point(0, 0);
    float flAngle = 0;
    Polygon poly = new Polygon();

    for( int ii=0 ; ii<5; ii++ ) {
        flAngle += Math.PI*4/5;
        pt.x = (int)(Math.cos(flAngle) * radius);
        pt.y = (int)(Math.sin(flAngle) * radius);
        poly.addPoint(pt.x, pt.y);
    }

    return poly;
}
```

## addPoint

**ClassName**

Polygon

**Purpose**

Adds a new vertex to the Polygon.

**Syntax**

public void addPoint(int x, int y);

**Parameters**

***int x***

The x coordinate of the vertex to add to the Polygon.

***int y***

The y coordinate of the vertex to add to the Polygon.

**Imports**

None.

**Description**

Appends a new vertex to the Polygon. The new vertex is the last in the list of vertices, and conceptually is connected by a line to the first vertex and the second-to-the-last vertex.

**Returns**

None.

**Example**

See the example for the Polygon class constructor above.

## getBoundingBox

**ClassName**

Polygon

**Purpose**

Gets the size of a bounding rectangle for this Polygon.

**Syntax**

public Rectangle getBoundingBox();

**Parameters**

None.

**Imports**

None.

**Description**

A Rectangle object which represents a rectangle that contains all the vertices of the Polygon. The Rectangle object which is returned is actually a private member variable of the Polygon object. This means that if you modify the member variable of the Rectangle object, and later call getBoundingBox, the changes you previously made to the Rectangle will be retained.

**Returns**

A Rectangle object which completely contains the Polygon.

**Example**

This example Component displays a Polygon. The Component uses the size of the Polygon's bounding rectangle as its preferred size.

```
public class MyPolyComponent extends Canvas {
    Polygon _poly = new Polygon();

    ...

    public Dimension preferredSize() {
        Dimension dim = new Dimension(
                _poly.getBoundingBox().width,
                _poly.getBoundingBox().height);
        return dim;
    }

    ...
}
```

## inside

**ClassName**

Polygon

**Purpose**

Tells whether a point lies within the Polygon.

**Syntax**

public boolean inside(int x, int y);

**Parameters**

***int x***

X coordinate of the point to test.

***int y***

Y coordinate of the point to test.

**Imports**

None.

**Description**

Tells whether or not a particular point lies within the Polygon. The evenodd insideness rule is used to determine whether or not the point is inside the Polygon's boundaries.

**Returns**

True is returned if the point lies inside the Polygon; otherwise false. The even-odd insideness rule is used to test for insideness.

## Rectangle

**Purpose**

A Rectangle object represents a rectangle on a two-dimensional surface.

**Syntax**

public class Rectangle

**Description**

Rectangles are represented by an origin point, which is the upper-left corner of the rectangle, a width, and a height. Of all the shape classes, the Rectangle is by far the most completely implemented. Several methods for working with, and manipulating, rectangles are implemented. For example, methods for computing the intersection and union of rectangles are included. Also included are methods for combining rectangles with Point and Dimension objects. Figure 7-14 shows the class hierarchy for the Rectangle class.

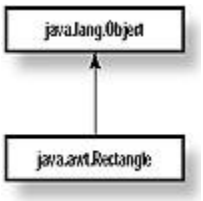


**Figure 7-14** Class hierarchy for the Rectangle class

**Package**

*java.awt*

**Imports**

None.

**Constructors**

public Rectangle();
public Rectangle(int x, int y, int width, int height);
public Rectangle(int width, int height);
public Rectangle(Point p, Dimension d);
public Rectangle(Point p);
public Rectangle(Dimension d);

These six overloaded constructors, taken as a group, allow you to specify none, one, or both of the Rectangle defining parameters: its points of origin and its dimensions. Note that the parameterless constructor actually does not initialize the Rectangle's member variables. If you use this constructor, be sure to zero-out the Rectangle using reshape.

**Parameters**

The following table lists the Rectangle class' public member variables.

| Member | Description |
|---|---|

| | |
|---|---|
| int *x* | The *x* and *y* parameters define the Rectangle's point of origin. |
| int *y* | |
| int *width* | The *width* and *height* parameters define the Rectangle's dimensions. |
| int *height* | |

## reshape

**ClassName**

Rectangle

**Purpose**

Used to change both the Rectangle's origin and dimensions.

**Syntax**

public void reshape(int x, int y, int width, int height);

**Parameters**

***int x***

The x coordinate of the new origin of the Rectangle.

***int y***

The y coordinate of the new origin of the Rectangle.

***int width***

The new width of the Rectangle.

***int height***

The new height of the Rectangle.

**Imports**

None.

**Descriptions**

Modifies the origin point, the width, and the height of the Rectangle. Note that negative values are allowed, as the example for this method shows.

**Returns**

None.

**See Also**

The resize method of the Rectangle class.

**Example**

The Rectangle class allows negative widths and heights. If a Rectangle's width or height is less than zero, then the origin point is no longer the upper-left corner of the rectangle. However, Components, which have a Rectangle of display surface, do not allow widths nor heights less than zero. To facilitate the use of negative widths and heights, the following method uses reshape to keep the origin as the upper-left corner of the rectangle.

```
public void fixDimension(Rectangle r) {
    int xNew, yNew, wNew, hNew;
    xNew = r.x + Math.min(r.width, 0);
    yNew = r.y + Math.min(r.height, 0);
    wNew = Math.abs(r.width);
    hNew = Math.abs(r.height);
```

```
    r.reshape(xNew, yNew, wNew, hNew);
}
```

## move

**ClassName**
Rectangle
**Purpose**
Changes the Rectangle's point of origin.
**Syntax**
public void move(int x, int y);
**Parameters**
***int x***
New x coordinate of the origin of the Rectangle.
***int y***
New y coordinate of the origin of the Rectangle.
**Imports**
None.
**Description**
Moves the Rectangle by changing its point of origin to another specified point.
**Returns**
None.
**Example**
See the example for the translate() method of the Rectangle class.

## translate

**ClassName**
Rectangle
**Purpose**
Moves the Rectangle's point of origin a specified distance along and X and Y axes.
**Syntax**
public void translate(int dx, int dy);
***int dx***
Distance along the X axis to move the origin of the Rectangle.
***int dy***
Distance along the Y axis to move the origin of the Rectangle.
**Imports**
None.
**Description**
Moves the Rectangle a specified distance along the X and Y axes. The origin after the call is distance (*dX*, *dY*) from the origin before the call.
**Example**

This example demonstrates how to implement translate as a call to move, since these two methods are so similar.

```
public void translate(Rectangle r, int dx, int dy) {
    // translate() is just a relative move()...
    r.move(r.x + dx, r.y + dy);
}
```

## resize

**ClassName**

Rectangle

**Purpose**

Modified the dimensions of the Rectangle.

**Syntax**

public void resize(int width, int height);

**Parameters**

***int width***

New width of the Rectangle.

***int height***

New height of the Rectangle.

**Imports**

None.

**Description**

Changes the width and height of the Rectangle without modifying the Rectangle's origin.

**Returns**

None.

**Example**

The CrossHairs component uses four rectangles to draw a cross-hair that centers wherever the user clicks the mouse.

```
public class CrossHairs extends Canvas {
    Rectangle _rNE, _rNW, _rSE, _rSW;

    public CrossHairs() {
        _rNE = Rectangle(0, 0, 0, 0);
        _rNW = Rectangle(0, 0, 0, 0);
        _rSE = Rectangle(0, 0, 0, 0);
        _rSW = Rectangle(0, 0, 0, 0);
    }

    // when component resized, resize rects
    public void resize(int width, int height) {
        super.resize(width, height);

        _rNW.reshape(0, 0, width/2, height/2);
        _rSW.reshape(0, height/2,
                width/2, height/2);

        _rNE.reshape(width/2, 0,
                width/2, height/2);
        _rSE.reshape(width/2, height/2, width/2,
```

```
                height/2);
        repaint();
    }

    public void paint(Graphics g) {
        g.drawRect(_rNW.x,_rNW.y,_rNW.width,_rNW.height);
        g.drawRect(_rNE.x,_rNE.y,_rNE.width,_rNE.height);
        g.drawRect(_rSW.x,_rSW.y,_rSW.width,_rSW.height);
        g.drawRect(_rSE.x,_rSE.y,_rSE.width,_rSE.height);
    }

    public boolean mouseDown(Event evt, int x, int y) {
        int width = size().width;
        int height = size().height;

        _rNE.resize(x, y);
        _rSE.reshape(0, y, x, height - y);
        _rNW.reshape(x, 0, width - x, y);
        _rSW.reshape(x, y, width - x, height - y);

        repaint();
        return true;
    }
}
```

## inside

**ClassName**

Tells whether a point is inside the Rectangle.

**Syntax**

public boolean inside(int *x*, int *y*);

**Parameters**

***int x***

The x coordinate of the point to test.

***int y***

The y coordinate of the point to test.

**Imports**

None.

**Description**

Tells whether or not the specified point lies inside the Rectangle. Points which fall on the border of the Rectangle are considered inside the Rectangle.

**Returns**

If the given point lies inside the Rectangle, or on its border, then true is returned. False is returned if the point lies outside the Rectangle.

**Example**

This example component draws a rectangle whose lower-right corner is the place where the last mouse click occurred.

```
public class GrowRect extends Canvas {
    Rectangle _r = new Rectangle(0, 0, 0, 0);

    public GrowRect() {}
```

```
    public void paint(Graphics g) {
        g.drawRect(_r.x, _r.y, _r.width, _r.height);
    }

    public boolean mouseDown(Event evt, int x, int y) {
        if(!_r.inside(x, y))
            _r.add(x, y);
        else
            _r.resize(x, y);
        repaint();
        return true;
    }
}
```

## intersects

**ClassName**

Rectangle

**Purpose**

Tells whether a Rectangle intersects with another.

**Syntax**

public boolean intersects(Rectangle r);

**Parameters**

***Rectangle r***

This external Rectangle is tested to see if it intersects this Rectangle.

**Imports**

None.

**Description**

Tells whether or not another Rectangle intersects this one. Rectangles which share a segment of border or a corner point are considered to intersect.

**Returns**

If the external Rectangle intersects this Rectangle, or the two Rectangles share a segment of border or a corner point, then the Rectangles are said to overlap. In this case, true is returned. Otherwise, false is returned.

**Example**

This example shows one possible implementation of the intersects method.

```
public class Rectangle {
...

    public boolean intersects(Rectangle r) {
        Rectangle r_intersect = intersection(r);
        return r_intersect.isEmpty();
    }

...
}
```

## intersection

**ClassName**

Rectangle

**Purpose**

Calculates the intersection of this Rectangle with another.

**Syntax**

public Rectangle intersection(Rectangle r);

**Parameters**

***Rectangle r***

The external Rectangle to intersect with this one.

**Imports**

None.

**Description**

Computes the new Rectangle which is the intersection of this Rectangle and an external Rectangle object. If the two Rectangles do not overlap, then the resulting intersection has a zero width and a zero height. The origin of this null-intersection Rectangle has an interesting relationship to the original two Rectangles, but it is essentially useless. Use isEmpty to test whether the resultant Rectangle represents an empty intersection.

**Returns**

A new Rectangle object is created and returned which lies completely within both the external Rectangle and this Rectangle.

**Example**

See the example for the intersect method of the Rectangle class.

## union

**ClassName**

Rectangle

**Purpose**

Finds a bounding Rectangle for this Rectangle and another.

**Syntax**

public Rectangle union(Rectangle r);

**Parameters**

***Rectangle r***

External Rectangle which is to be unioned with this Rectangle.

**Imports**

None.

**Description**

Computes a new Rectangle which completely contains this Rectangle and an external Rectangle object. The resultant union Rectangle is the smallest Rectangle which can contain both input rectangles.

**Returns**

The returned Rectangle is a new Rectangle which is the smallest Rectangle which can contain the two input Rectangles.

## add

**ClassName**

Rectangle

**Purpose**

Unioning two Rectangles, or a Rectangle and a Point.

**Syntax**

public void add(int x, int y); public void add(Point pt); public void add(Rectangle r);

**Parameters**

***int x***

The x coordinate of the point to add to this Rectangle.

***int y***

The y coordinate of the point to add to this Rectangle.

***Point pt***

Point to add to this Rectangle.

***Rectangle r***

Rectangle to add to this Rectangle.

**Imports**

None.

**Description**

The addition of a Rectangle and a point is analogous to the union operation. The new dimension of this Rectangle is the smallest which can contain both the original Rectangle and a given point. The addition of another Rectangle uses the same procedure as a union operation. The overloaded add methods modify the origin, width, and height of this Rectangle. The resultant dimensions and placement of this Rectangle represent the smallest Rectangle which can contain both the original Rectangle and any external point or external Rectangle object.

**Example**

See the example for the inside method of the Rectangle class.

## grow

**ClassName**

Rectangle

**Purpose**

Grows a Rectangle’s Dimensions.

**Syntax**

public void grow(int h, int v);

**Parameters**

***int h***

Distance in the horizonal direction to move the left and right borders away from the center of the Rectangle. Negative numbers move the borders toward the center.

***int v***

Distance in the vertical direction to move the top and bottom borders away from the center of the Rectangle. Negative numbers move the borders toward the center.

**Imports**

None.

**Description**

Makes the dimensions of the Rectangle larger (or smaller) according to the magnitude of the horizontal and vertical grow parameters. The Rectangle is modified in such a way that all broders are moved a uniform distance away from (or towards) the center of the original Rectangle.

**Returns**

None.

**Example**

This example demonstrates how the grow method can be used to shrink rectangles, too.

```
public void shrink(Rectangle r, int h, int v) {
    r.grow(-h, -v);
}
```

### isEmpty

**ClassName**

Rectangle

**Purpose**

Tells whether either of the Rectangle's Dimensions are 0.

**Syntax**

public boolean isEmpty();

**Parameters**

None.

**Imports**

None.

**Description**

Tests the Rectangle to see if it has a non-zero volume. That is, whether or not both the width and height parameters are non-zero. It is useful to use isEmpty() to see if the results of the intersection() method produce a non-empty Rectangle object.

**Returns**

True is returned if the Rectangle indeed has a zero internal volume. This indicates either the width or the height of the Rectangle is zero (or both).

**Example**

See the example for the intersects method of the Rectangle class.

## The Graphical Object Project: Doodle

The Graphical object SuperBible Project is the Doodle application. Doodle is a simple whiteboard application. It allows you to draw Lines, Rectangles, Polygons, and text on the drawing surface. Through simple mouse clicks and drags, you define the shape and placement of each "doodle" on the whiteboard surface. Figure 7-15 shows the Doodle

application in action. The Java code for the Doodle application is included under the directory Chap07 on the CD accompanying this book.

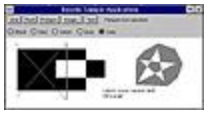

**Figure 7-15** Screenshot of the Doodle application running

Doodle's user-interface employs two toolbars: shape and color. The shape toolbar has five buttons, labeled Line, Rect, Polygon, Image..., and Text. You click on the button for the type of doodle you want to draw. The Color toolbar contains a set of several radio buttons. When you select a radio button, all doodles will be drawn using the corresponding Color.

Doodle employs "virtual Components" to draw each of the doodles you define. For example, instances of the DoodleLine class, also defined in Doodle.java, draw a single line on the whiteboard surface. DoodlePoly objects draw a polygon; DoodleRect objects draw filled rectangles, etc.

A virtual Component is a Canvas-derived class which explicitly prevents its peer from being created. Why stop the peer from being created? Isn't the whole point of Components that it allows you to draw on a rectangle of the on-screen desktop? The reason is that overlapping Components, created with a peer, automatically "clip" siblings. That is, Components added to the same Container cover each other, so overlapping sibling Components are not transparent. Imagine drawing two diagonal lines right next to each other: If each line was represented by a single DoodleLine Component, then the Component on top would completely overwrite the Component on the bottom. Doodle was created with transparent Components to prevent this overwriting problem.

By removing the operating system peer, each Doodle graphical object Component represents a rectangle of the whiteboard's surface. The whiteboard uses each Doodle Component's paint method to render the doodle on its rectangle, but the Doodle Component makes sure it does not erase anything. The practical result is transparent Component objects.

But removing a Doodle Component's peer has two important side effects:

- No paint() calls are made by the Java runtime system for the Component. Instead, the whiteboard, which contains the Doodle Components, must explicitly call each Component's paint() method to let the Doodles draw themselves.
- No Events, such as mouse or keyboard Events, are delivered by the Java runtime system to the Doodle Component. Again, the whiteboard must explicitly deliver these Events to the correct Component object.

The Doodle application overcomes both of these shortcomings to allow each virtual Component to act like a Canvas with a peer, in addition to being transparent.

## Assembling the Project

Follow these steps to assemble the Doodle application.

**1.** Create a text file named Doodle.java. Make sure the correct packages are imported and implement the static method, which just creates Doodle's main Frame window. Here is the code:

```
import java.awt.*;
import java.util.Hashtable;
import java.util.Vector;
import java.awt.image.ImageObserver;
public class Doodle {
    public static void main(String[ ] astrArgs) {
        // Create main window, which is a DoodleFrame.
        // DoodleFrame constructor creates its own
        // child windows.
        DoodleFrame df =
            new DoodleFrame( "Doodle Sample Application" );
        df.resize(500, 300);
        df.show();
        return;
    }
}
```

**2.** Create the Doodle application's main Frame class. This class manages the shape and color toolbars. Start by declaring the member variables for these toolbars. Also declare the DoodleView. The DoodleView is the actual whiteboard on which we will draw the doodles.

```
class DoodleFrame extends Frame {
    // Doodle type buttons, which will live
    // in a toolbar panel. Panel is created
    // and filled with the buttons here before
    // being added as a Component of this Frame.
    // Frame keeps track of last button pressed.
    // A label reports which was the last button
    // pressed.
    Button _buttonLastPressed = null;
    Button _buttonLine = new Button( "Line" );
    Button _buttonRect = new Button( "Rect" );
    Button _buttonPoly = new Button( "Polygon" );
    Button _buttonText = new Button( "Text" );
    Label _labelTool = new Label( "No tool selected" );
    Panel _panelToolbar = new Panel();

    // There is also a Colorbar, which allows the
    // user to select from among several Colors
    // to paint their doodles. A Hashtable associating
    // checkbox to a Color object is also maintained
    // so Doodle can find which Color to use.
    Checkbox _checkBlack = new Checkbox( "Black" );
    Checkbox _checkRed = new Checkbox( "Red" );
    Checkbox _checkGreen = new Checkbox( "Green" );
    Checkbox _checkBlue = new Checkbox( "Blue" );
```

```
    Checkbox _checkGray = new Checkbox( "Gray" );
    Panel _panelColorbar = new Panel();
    Hashtable _hashCheckToColor = new Hashtable();

    // The DoodleView keeps track of and displays
    // all the doodles the user has drawn so far.
    DoodleView _DoodleView = null;}
```

**3.** The DoodleFrame constructor creates the interface for the Doodle application. It creates the two toolbars and adds the DoodleView to the Frame. The DoodleFrame uses a GridBagLayout to arrange the two toolbars and the DoodleView. See Chapter 6 for an explanation of the LayoutManager classes. In this application, the GridBagLayout ensures the toolbars and the DoodleView arrange themselves neatly no matter what the size of the DoodleFrame. Here is the code for the constructor:

```
public DoodleFrame(String strTitle) {
    super(strTitle);

    // This Frame uses a GridBagLayout to manage
    // two Components: the toolbar Panel will
    // sit flush left, taking about 20 percent of the
    // height of this Frame. The DoodleView will
    // sit flush right, taking about 80 percent of the
    // height of this Frame.
    GridBagLayout gbl = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.fill = GridBagConstraints.BOTH;
    gbc.weightx = 1.0;

    setLayout(gbl);

    // Add buttons (created as part of object initialization)
    // to the toolbar panel.
    _panelToolbar.setLayout(
           new FlowLayout(FlowLayout.LEFT));
    _panelToolbar.add( _buttonLine );
    _panelToolbar.add( _buttonRect );
    _panelToolbar.add( _buttonPoly );
    _panelToolbar.add( _buttonText );
    _panelToolbar.add( _labelTool );

    // Add toolbar Panel to this Frame, making
    // the gridheight = 1.
    gbl.setConstraints(_panelToolbar, gbc);
    add(_panelToolbar);

    // Add Colorbar panel to this frame after
    // adding all the checks to it.
    CheckboxGroup cbg = new CheckboxGroup();
    _checkBlack.setCheckboxGroup(cbg);
    _checkBlack.setState(true);
    _checkRed.setCheckboxGroup(cbg);
    _checkGreen.setCheckboxGroup(cbg);
    _checkBlue.setCheckboxGroup(cbg);
    _checkGray.setCheckboxGroup(cbg);
```

```
    _panelColorbar.setLayout(new FlowLayout(FlowLayout.LEFT));
    _panelColorbar.add(_checkBlack);
    _panelColorbar.add(_checkRed);
    _panelColorbar.add(_checkGreen);
    _panelColorbar.add(_checkBlue);
    _panelColorbar.add(_checkGray);

    gbc.gridy = 1;
    gbl.setConstraints(_panelColorbar, gbc);
    add(_panelColorbar);

    // Associate each color checkbox with a Color to use
    // when that checkbox is selected.
    _hashCheckToColor.put(_checkBlack, Color.black);
    _hashCheckToColor.put(_checkRed, Color.red);
    _hashCheckToColor.put(_checkGreen, Color.green);
    _hashCheckToColor.put(_checkBlue, Color.blue);
    _hashCheckToColor.put(_checkGray, Color.gray);

    // Create the DoodleView. Add it to the
    // grid bag layout with gridwidth = 4.
    _DoodleView = new DoodleView(this);
    gbc.gridy = 2;
    gbc.weighty = 4.0;
    gbl.setConstraints(_DoodleView, gbc);
    add(_DoodleView);

    return;
}
```

**4.** The DoodleFrame must catch WINDOW_DESTROY Events so it knows when to destroy the main window. Here's the code for DoodleFrame's overridden handleEvent method:

```
public boolean handleEvent(Event evt) {
       // The WINDOW_DESTROY event must be
       // handled by disposing of this Frame.
       if( (evt.target == this) &&
                 (evt.id == Event.WINDOW_DESTROY)) {
           dispose();
           System.exit(0);
           return true;
       }

       return super.handleEvent(evt);
}
```

**5.** When a toolbar button is pressed, the DoodleFrame catches the event and "remembers" (in a member variable) the last button pressed. The buttons indicate what type of doodle the user wants to paint. Here is the code for DoodleFrame's overridden action implementation:

```
// action() will be called whenever one of the
// toolbar buttons is pressed. When this happens,
// report new doodle tool and store reference
// to the button pressed.
public boolean action(Event evt, Object what) {
    // Handle toolbar panel buttons...
    if(evt.target instanceof Button) {
```

```
            _buttonLastPressed = (Button)evt.target;
            _labelTool.setText(_buttonLastPressed.getLabel()
                   + " tool selected");

                // Tell the DoodleView to abandon the active
                // doodle if its incomplete.
                _DoodleView.notifyToolChanged();

                _panelToolbar.invalidate();
                _panelToolbar.validate();
                return true;
            }

            // Everything else can be ignored.
            return false;
        }
```

**6.** The DoodleFrame.createDoodleCompInstance method creates a new Doodle component ("DoodleComp") of the type indicated by the last toolbar button pressed. The DoodleView calls this method when it detects the user has started drawing a new doodle. Here's the code for that method:

```
// createDoodleCompInstance() creates a new DoodleComp
// of the type indicated by the last toolbar button
// that was clicked. This is called by the DoodleView
// when it determines its time to create a new doodle.
public DoodleComp createDoodleCompInstance() {
    // return null if no tool is selected.
    if(null == _buttonLastPressed)
        return null;

     // according to which was last button pressed,
     // make corresponding DoodleComp object.
     if(_buttonLine == _buttonLastPressed)
        return new DoodleLine(getDoodleColor());
     if(_buttonRect == _buttonLastPressed)
        return new DoodleRect(getDoodleColor());
     if(_buttonPoly == _buttonLastPressed)
        return new DoodlePoly(getDoodleColor());
     if(_buttonText == _buttonLastPressed)
        return new DoodleText(getDoodleColor(),
                 getFont());

     return null;
}
```

**7.** The other thing the DoodleView needs to get from the DoodleFrame is the color currently selected in the color toolbar. The DoodleFrame.getDoodleColor method returns the Color corresponding to the color selected in the color toolbar. Here's the code for that method:

```
// getDoodleColor returns the hashtable entry for the
// colorbar checkbox which is currently checked.
public Color getDoodleColor() {
    Checkbox checkCurrent = null;

    Checkbox[ ] acheckList = new Checkbox[5];
    acheckList[0] = _checkBlack;
    acheckList[1] = _checkRed;
```

```
    acheckList[2] = _checkGreen;
    acheckList[3] = _checkBlue;
    acheckList[4] = _checkGray;

    for( int i=0 ; i<acheckList.length ; i++ )
        if(true == acheckList[i].getState())
            checkCurrent = acheckList[i];

    if(null == checkCurrent)
        return null;

    return (Color)_hashCheckToColor.get(checkCurrent);
}
```

**8.** That was the last DoodleFrame method. The DoodleFrame class is now fully implemented. Now we declare the DoodleView class. The DoodleView is the whiteboard panel which is the parent of all the DoodleComp virtual Components. The DoodleFrame uses double-buffering so that redrawing of the screen is smooth. Here's the declaration of the DoodleView class and its member variables:

```
class DoodleView extends Panel {
    // The panel keeps track of the Doodle Frame
    // so it can query it for current doodling
    // tool.
    DoodleFrame _DoodleFrame = null;
    DoodleComp _ActiveDoodleComp = null;

    // To make visual updating flicker-free, use
    // a permanent in-memory double-buffer.
    Image _img2blBuf = null;
    Graphics _g2blBuf = null;
}
```

**9.** The DoodleView constructor requires a reference to the application's DoodleFrame to be passed. In the constructor, the DoodleView does some minor initialization work. Here is the code for the DoodleView constructor:

```
public DoodleView(DoodleFrame df) {
    _DoodleFrame = df;

    // Background of the DoodlePanel is
    // always white, painting color is
    // always black. (All painting will
    // be done in XOR mode with alternate
    // color white.)
    setBackground( Color.white );
    setForeground( Color.black );

    // Use a NullLayout, which will just
    // leave all Components alone in place.
    setLayout(new NullLayout());
}
```

**10.** To avoid "flicker" caused by the default implementation of update, DoodleView must override update to simply call paint.

```
public void update(Graphics g) {
    paint(g);
}
```

**11.** To implement double-buffering, the DoodleView must keep around an in-memory Image and associated Graphics object. The Image must be exactly the same size as the DoodleView itself. To achieve this, DoodleView.reshape is implemented. The reshape method is automatically called whenever the DoodleView is resized. So in our implementation, DoodleView calls the base class implementation of reshape, but also creates a new Image and associated Graphics of exactly the correct size. Here's the code:

```
// Whenever the view is reshaped, must re-allocate
// our in-memory image to be the same size.
public void reshape(int x, int y, int width, int height) {
    super.reshape(x, y, width, height);
    _img2blBuf = createImage(size().width, size().height);

    // Make sure to dispose of unused Graphics.
    if(null != _g2blBuf)
        _g2blBuf.dispose();
    _g2blBuf = _img2blBuf.getGraphics();
}
```

**12.** Since DoodleComps have no peers, the Java system never tells them to repaint themselves. Instead, the DoodleView will explicitly tell each DoodleComp to paint itself onto the in-memory Image whenever the DoodleView's paint method is called. Here is the code for DoodleView's paint method:

```
// Since all DoodleComps have no peers, this
// DoodleView is responsible for making sure
// they paint themselves. Use double-buffering
// to minimize visual flicker.
public void paint(Graphics g) {
    _g2blBuf.setColor(getBackground());
    _g2blBuf.fillRect(0, 0, size().width,
            size().height);
    _g2blBuf.setFont(getFont());
    _g2blBuf.setColor(getForeground());

    for( int i=0 ; i<countComponents() ; i++ ) {
        Component c = getComponent(i);
        Rectangle r = c.bounds();
        if(true == r.intersects(g.getClipRect())) {
            Graphics gComp =
                    _g2blBuf.create(r.x, r.y, r.width,
                            r.height);

            c.paint(gComp);
            gComp.dispose();
        }
    }

    // Copy in-memory image to Graphics g.
    g.drawImage(_img2blBuf, 0, 0, this);

    return;
}
```

**13.** The DoodleFrame calls the DoodleView's notifyToolChanged method whenever the user selects a new shape button on the shape toolbar. This gives the

DoodleView a chance to abort any doodle which is only half-drawn. For example, let's say the user was typing some text. The Doodle application requires that the user clicks with the mouse to notify the application when she is done with a particular doodle. But instead the user clicks on a different tool on the shape toolbar. In that case, the DoodleView should remove the half-finished DoodleComp. That's what notifyToolChanged does. Here's the code:

```
// The DoodleFrame calls notifyToolChanged  whenever
// the user selects a doodle tool button. The view
// abandons any incomplete doodles and gets rid
// of them if this happens.
public void notifyToolChanged() {
    if(null != _ActiveDoodleComp) {
        remove(_ActiveDoodleComp);
        _ActiveDoodleComp = null;
    }
    return;
}
```

**14.** Interpretation of mouse and keyboard events is left completely up to the DoodleComps themselves. The DoodleView passes all mouse and keyboard actions to the DoodleComp currently being edited, if there is one. The DoodleComp lets the DoodleView know when it is done by either returning false from the mouse or keyboard Event, or by returning true from its isComplete method at any time. Here's DoodleView's mouse Event methods, which pass the Events on to the active Doodle:

```
// If a mouse-down occurs, and there is no active
// doodle, then a new one is created using
// DoodleFrame.createDoodleCompInstance().
public boolean mouseDown(Event evt, int x, int y) {
    if(null == _ActiveDoodleComp) {
        _ActiveDoodleComp =
                _DoodleFrame.createDoodleCompInstance();
        if(null == _ActiveDoodleComp)
            return false;
        add(_ActiveDoodleComp);
    }

    // Pass mouse message on to the active doodle.
    boolean fRet = _ActiveDoodleComp.mouseDown(evt, x, y);

    if((false==fRet) ||
            (true==_ActiveDoodleComp.isComplete()))
        _ActiveDoodleComp = null;

    return fRet;
}

// Mouse drags are simply passed on to the active
// doodle, if there is one. If active doodle
// isComplete() after processing the message, then
// we can drop reference.
public boolean mouseDrag(Event evt, int x, int y) {
    if(null == _ActiveDoodleComp)
        return false;
```

```
    boolean fRet =
            _ActiveDoodleComp.mouseDrag(evt, x, y);

    if((false==fRet) ||
            (true==_ActiveDoodleComp.isComplete()))
        _ActiveDoodleComp = null;

    return fRet;
}

// Mouse ups are simply passed on to the active doodle,
// if there is one. If active doodle isComplete() after
// processing the message, then we can drop reference.
public boolean mouseUp(Event evt, int x, int y) {
    if(null == _ActiveDoodleComp)
        return false;

    boolean fRet =
            _ActiveDoodleComp.mouseUp(evt, x, y);

    if((false==fRet) ||
            (true==_ActiveDoodleComp.isComplete()))
        _ActiveDoodleComp = null;

    return fRet;
}
```

**15.** Keyboard Events, specifically keyDown events, are treated the same as mouse Events by the DoodleView. Here's the code for the DoodleView's keyDown Event handler.

```
// key presses are simply passed on to the active
// doodle, if there is one. If active doodle
// isComplete() after processing the message,
// then we can drop reference.
public boolean keyDown(Event evt, int key) {
    if(null == _ActiveDoodleComp)
        return false;

    boolean fRet =
            _ActiveDoodleComp.keyDown(evt, key);

    if((false==fRet) ||
            (true==_ActiveDoodleComp.isComplete()))
        _ActiveDoodleComp = null;

    return fRet;
}
```

**16.** That's it for the DoodleView. Now for the individual DoodleComp classes. Each DoodleComp represents, and is able to edit and draw a single doodle, such as a line or a polygon. All types of DoodleComps are derived from the DoodleComp class, which does nothing more than implement a few methods which will be re-used by the individual doodle classes, and declare the isComplete method. Doodles being edited let the DoodleView know they do not want any more events passed to them by returning true from the isComplete method. Here's the full implementation of the DoodleView class:

```
abstract class DoodleComp extends Canvas {
    public DoodleComp(Color c) {
        setForeground(c);
    }

    // Avoid adding a peer by overriding addNotify().
    public void addNotify() {
        return;
    }
    // paint() is called explicitly by the Doodle-
    // View when it is painting itself. Graphics
    // has no guarantee of state, so this implementation
    // does a little initialization. Derived classes
    // can call super.paint() to have this code run
    // before doing any actual rendering on the
    // Graphics.
    public void paint(Graphics g) {
        g.setColor(getForeground());
        g.setFont(getFont());
        g.setXORMode( Color.white );
        return;
    }

    // isComplete() returns true if the doodle
    // has been completed.
    public abstract boolean isComplete();
}
```

**17.** Now for the individual doodles. The simplest is the DoodleLine. The DoodleLine just waits for the mouse button to be let go. At that point, the Line is defined by two points: where the mouse was when the DoodleLine was created, where the mouse was when the user let go of the mouse button. Here's the code for the DoodleLine's constructor, member variables, and isComplete method implementation:

```
class DoodleLine extends DoodleComp {
    Point _pt1 = null;
    Point _pt2 = null;
    Point _ptDrag = null;

    // Must be constructed with a color.
    public DoodleLine(Color c) {
        super(c);
    }

    public boolean isComplete() {
        if((null != _pt1) && (null != _pt2))
            return true;
        return false;
    }
```

**18.** The DoodleLine keeps track of where the user is dragging the mouse. This is important because if the user drags the mouse to the left or upward of the upper-left corner of the DoodleLine, then the DoodleLine must resize and reposition itself with its origin far enough towards the application's upper-left corner to encompass the new point. Here's the code for DoodleLine.mouseDrag:

```
public boolean mouseDrag(Event evt, int x, int y) {
```

```
    if(isComplete())
        return false;

    Point pt = location();
    _ptDrag = new Point(x-pt.x, y-pt.y);

    if(_ptDrag.x < 0) {
        _pt1.x += -_ptDrag.x;
        _ptDrag.x = 0;
    }
    if(_ptDrag.y < 0) {
        _pt1.y += -_ptDrag.y;
        _ptDrag.y = 0;
    }

    Rectangle r = bounds();
    r.add(new Point(x, y));
    reshape(r.x, r.y, r.width, r.height);

    getParent().repaint(r.x, r.y, r.width, r.height);

    return true;
}
```

**19.** The initial MOUSE_DOWN Event defines where the first point of the DoodleLine is, and a MOUSE_UP Event causes the DoodleLine to report that it is done editing. Here's the code for the mouseDown and mouseUp event handlers of the DoodleLine class:

```
public boolean mouseDown(Event evt, int x, int y) {
    if(isComplete())
            return false;

    move(x, y);
    _pt1 = new Point(0, 0);

    Rectangle r = bounds();
    getParent().repaint(r.x, r.y, r.width, r.height);

    return true;
}

public boolean mouseUp(Event evt, int x, int y) {
    if(isComplete())
        return false;

    Point pt = location();
    _pt2 = new Point(x-pt.x, y-pt.y);
    if(_pt2.x < 0) {
        _pt1.x += -_pt2.x;
        _pt2.x = 0;
    }
    if(_pt2.y < 0) {
        _pt1.y += -_pt2.y;
        _pt2.y = 0;
    }

    Rectangle r = bounds();
```

```
    r.add(new Point(x, y));
    reshape(r.x, r.y, r.width, r.height);

    getParent().repaint(r.x, r.y, r.width, r.height);

    return true;
}
```

**20.** The paint method of all DoodleComps is called explicitly by the DoodleView from its paint implementation. Here is DoodleLine's implementation, which makes sure to call its superclass implementation of paint first (DoodleComp.paint). The superclass implementation sets up the Graphics with the proper colors, fonts, and painting mode. Here's DoodleLine's painting code:

```
public void paint(Graphics g) {
    super.paint(g);

    if(null == _pt1)
        return;

    if(isComplete()) {
        g.drawLine(_pt1.x, _pt1.y, _pt2.x, _pt2.y);
        return;
    }

    if(null != _ptDrag)
        g.drawLine(_pt1.x, _pt1.y, _ptDrag.x,
                   _ptDrag.y);
    return;
}
```

The rest of the DoodleComp classes are very similar to DoodleLine. Similarly to DoodleLine, they handle mouse and keyboard events until the user is done editing the doodle. They all use pretty much the same code to ensure the bounding rectangle of the DoodleComp continually contains the location of the mouse.

## How It Works

The following table lists all the classes defined in the Doodle application, and a short description of each.

| Class | Description |
|---|---|
| Doodle | The class contains the static main() method for the Doodle application. This is the only public class in the Doodle project. |
| DoodleFrame | The main window of the Doodle application. The DoodleFrame window creates the two toolbars, allowing the user to select a type of doodle to draw on the whiteboard surface and a Color to draw in. The two toolbars and the whiteboard (DoodleView) are the three Components added to this Container. |

| | |
|---|---|
| DoodleView | This is the whiteboard Component. This is where the guts of making the DoodleComp (doodle Component) virtual Components work. Mouse and keyboard Events on the whiteboard are delivered explicitly to the correct DoodleComp. When the whiteboard is supposed to paint itself, it explicitly calls each DoodleComp's paint() method with a unique Graphics attached to the DoodleComp's rectangle. |
| DoodleComp | This is the base class for all the doodle Component objects. The methods, which are implemented the same for each doodle Component, are implemented in this class. |
| DoodleLine | Each doodle line is a single line the user drew on the whiteboard surface. It collects a mouseDown() and a mouseUp() Event call (sent explicitly by the whiteboard) to define a line to draw. |
| DoodleRect | Each doodle rectangle is a filled rectangle the user drew on the whiteboard surface. It collects a mouseDown() and mouseUp() Event call to define the upper-left and lower-right corners of the rectangle. |
| DoodlePoly | Each doodle polygon is a filled Polygon the user drew on the whiteboard surface. The user defines the polygon by a series of mouse down,drag, and mouse up actions. |
| DoodleText | Each String of text the user defined to be drawn on the whiteboard surface is stored as a DoodleText. All keyboard Events sent by the DoodleView to a DoodleText are stored up by the DoodleText. A mouse click defines the upper-right corner of the DoodleText's surface rectangle. |
| NullLayout | The doodle Components should be placed and sized exactly how the user defines them with her mouse clicks and drags. The whiteboard uses a NullLayout as its LayoutManager. NullLayout does absolutely nothing to the size or position of the doodle Components. Thus, the doodle Components can place themselves wherever they want using their own move(), resize(), or reshape() methods. |
| PolygonEx | There is no translate() method for the Polygon class in the Java API. The PolygonEx class defines a method to move, or translate, all the vertices in a Polygon object any distance in either the X or Y directions. |

---

### Use of Rectangles in Doodle

Each doodle object in the Doodle application is represented by a Component object. One drawback with the Component class is that it does not allow you to specify a negative width or height when reshaping the Component on the screen. For the Doodle application, this poses a little problem. While allowing the user to click-and-drag to

define a doodle, say for instance a LineDoodle, the Doodle application must continually reshape the LineDoodle component. If the user drags the mouse to the right or below the LineDoodle's rectangle, then the Doodle application simply adds to the width and/or height of the LineDoodle Component and redraws it. If, on the other hand, the user drags the mouse above or to the left of the LineDoodle component, both the origin and dimensions of the Component must be changed. The trick is to ensure that the upper-left-most extent of the Component is always the origin.

The code DoodleLine uses whenever it receives a mouse-drag is:

```
public boolean mouseDrag(Event evt, int x, int y) {
    // If line has already been completed
    // (ie, mouseUp occurred)
    // then ignore this mouseDrag event.
    if(isComplete())
            return false;

        // x and y parameters are relative to Container's
        // origin. Get Point relative to this Component's
        // origin in _ptDrag.
        Point pt = location();
        _ptDrag = new Point(x-pt.x, y-pt.y);

        // If _ptDrag is to the left or above this Component,
        // then change the origin of this component to
        // _ptDrag, and change _ptDrag to (0,0).
        if(_ptDrag.x < 0) {
            _pt1.x += -_ptDrag.x;
            _ptDrag.x = 0;
        }
        if(_ptDrag.y < 0) {
            _pt1.y += -_ptDrag.y;
            _ptDrag.y = 0;
        }

        // Reshape this Component to encompass the
        // new mouseDrag point.
        Rectangle r = bounds();
        r.add(new Point(x, y));
        reshape(r.x, r.y, r.width, r.height);

        // Since line has changed, repaint it by repainting
        // its rectangle within the Container's surface.
        getParent().repaint(r.x, r.y, r.width, r.height);

        // This Component will continue processing events, so
        // return true to indicate line is not yet complete.
        return true;
}
```

The DoodleRect and DoodlePoly virtual Components use a similar technique to reshape themselves whenever the user expands them to the left or upward. DoodleRect uses almost exactly the same code as DoodleLine. The big difference between the two is in the

paint method. DoodleLine paints a line between its internally stored *_pt1* and *_pt2* Point member variables, while DoodleRect fills a Rectangle which is its *bounds*.

The DoodlePoly class is different than DoodleLine or DoodleRect in that the user does not use a single mouse click-and-drag to define the Polygon. Instead, the user uses a series of mouse clicks-and-drags, one for each segment of the Polygon to draw. Instead of using mouseDrag to reshape the Polygon and redraw it, DoodlePoly uses mouseUp. Nevertheless, the technique is still the same: If the MOUSE_UP event is above or to the left of the DoodlePoly Component, then the DoodlePoly is moved and reshaped to keep the origin as the upper-left corner of the Component to avoid having a negative width or height.

# Chapter 8
# AWT Image Processing

Several utility classes and interfaces are included in the Java API for image processing. Using these APIs, Image objects can be created from raw data, the raw data of an existing Image object can be examined, and filters can be used to make modified versions of existing images. Image objects that you create from raw data or using filters can be used in exactly the same ways as Image objects created by the Java runtime system.

In the Java language, Image data is presented in a consistent format by objects which implement the ImageProducer interface. Image data is used by objects which implement the ImageConsumer interface. The ImageProducer and ImageConsumer interfaces are the basis for image processing in the Abstract Windows Toolkit (AWT) of Java.

This chapter details how ImageProducers, ImageConsumers, and ImageFilters work, and how you use them to create and modify images in Java. First, we will see how image pixel data is represented using arrays of bytes or integers in Java, and how the ColorModel object is used to interpret image pixel data. Next, we will see how the ImageProducer interface passes image pixel and ColorModel information to ImageConsumers. Finally, this chapter explains the ImageFilter, which is really both an ImageConsumer and an ImageProducer.

The project for this chapter, “MultiFilter,” is an extensible image filtering application. It allows the user to apply filters to arbitrary sections of Images. MultiFilter is designed to be extensible—after you see how it is designed and built, you will be able to drop in and test any ImageFilter you create.

## Image Data in Java

As with most computer representation of images, images in AWT are conceptually a two-dimensional array of pixels. Each pixel has color data associated with it. According to the

color storage model for the Image, this data can be stored in either a single 8-bit byte, or in a 32-bit integer.

The default color model stores eight bits each of red, green, blue, and “alpha” or “transparency” information packed into a single integer. This is the RGBα color model. Figure 8-1 illustrates the packing of RGBα data into a single integer. All Image objects have an associated ColorModel object capable of converting the specific Image’s color data into an RGBα representation. The ColorModel class encapulates the methods necessary to convert the Image pixel data for single pixels into the default RGBα packed bits format. Whenever raw Image pixel data is passed to a method, the ColorModel for that pixel data accompanies it. Sometimes the ColorModel is assumed to be the default RGBα model, such as in the RGBImageFilter.filterRGB method, which is passed RGBα pixel data but no ColorModel.

**Figure 8-1** RGBα color model

The ColorModel class is an abstract class that defines the methods which must be implemented by actual ColorModels. The two ColorModel classes included with the AWT are IndexColorModel and the DirectColorModel, both of which are also summarized in this chapter.

For example, an Image may use an IndexColorModel, where each pixel’s single byte of data is actually an index into a color table of 256 RGBα integers. Given a pixel’s 8-bit index into its color table, the IndexColorModel for the Image can supply the specific RGBα data for that pixel. All ColorModels implement getRGB to supply RGBα information for all pixels in the Image. This listing uses getRGB to retrieve RGBα information associated with pixel value 0.

```
URL urlImage;

//…
// urlImage is loaded with the URL for an image.
//…

Image img = Toolkit.getDefaultToolkit().getImage( urlImage );

//…
// Image data is allowed to be downloaded, for instance by a
// MediaTracker object.
//…

// Pass to getRBG() the native color data for a particular
// pixel. Below, native data '0’ is passed in, which could
// refer to the first entry in an indexed color table.
ColorModel cm = img.getColorModel();
int rgb = cm.getRGB(0);
```

The other ColorModel included in AWT is the DirectColorModel. The DirectColorModel can store packed bits of red, green, blue, and alpha values using arbitrary bitmasks for each of the color component fields in an integer. The width of the bitfields for each color component is completely configurable. The DirectColorModel internally computes the method for converting data stored using arbitrary bitfields into default RGB data. Like all ColorModel-derived classes, the DirectColorModel implements getRed, getGreen, getBlue, getAlpha, and getRGB.

For example, you can define a DirectColorModel that stores 12 bits of red data in the 12 highest bits, 4 bits of green data in the bottom 4 bits, 16 bits of blue data in the middle 16 bits, and no alpha data at all. The DirectColorModel can automatically convert this kind of pixel data into the default RGBα representation.

There are many other ColorModels which could be implemented, each best suited to a particular real-world application. For example, television images are encoded in the so-called "YIQ" or "luminescence/chromaticity" color model. One could define a YIQColorModel so that Images storing pixel data using the YIQ schema could be rendered on display surfaces by Java.

### Passing Image Data: The ImageProducer and ImageConsumer Interfaces

The AWT creates Image objects from data provided by ImageProducers. An ImageProducer delivers Image pixel data to ImageConsumers through the member methods of that interface. There are three types of ImageProducers implemented in the Java API:

- The Toolkit has built-in ImageProducers which can create Images from files stored on the local file system, or accessed through the network. The Toolkit's createImage method takes Image data and an associated ColorModel from an ImageProducer, and transfers it into structures appropriate for rendering the Image in a particular operating system.
- Any Image object can provide an ImageProducer which will deliver the Image's data to ImageConsumers. Image.getSource returns an ImageProducer that delivers the pixel data to ImageConsumer objects.
- The MemoryImageSource class is an ImageProducer which can deliver image data taken from an array of pixel data in memory.

You can create your own ImageProducers to accomplish additional Image creation tasks. For example, you may want to implement an ImageProducer to create images from lesser-known or unsupported graphic format files, like PostScript. To do so, just create a class which implements the ImageProducer interface and delivers pixel data to consumers through the member methods of the ImageConsumer interface.

ImageProducers require one or more ImageConsumer objects to accept image data. Figure 8-2 illustrates this relationship as a faucet and one or more buckets. The

ImageProducer funnels its image data to ImageConsumers which have been associated with the producer.

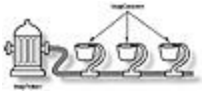

**Figure 8-2** The flow of Image pixel data from ImageProducer to multiple ImageConsumers

The ImageConsumer interface is implemented by objects wishing to process pixel data. The ImageProducer passes pixel data in one or, more usually, multiple calls to the ImageConsumer's setPixels:

```
class myConsumer extends ImageConsumer {
    public void setPixels(int x, int y, int width, int height,
            ColorModel model, int pixels[], int off, int scansize) {
        // do something with the 32-bit pixel data.
    }

    public void setPixels(int x, int y, int width, int height,
            ColorModel model, byte pixels[], int off, int scansize) {
        // do something with the 8-bit pixel data.
    }
}
```

Two overloaded versions of *setPixels* must be implemented, one to accept 8-bit pixel data, and the other to accept 32-bit pixel data. A rectangle of pixels from the Image is passed to setPixels. This rectangle might be the entire Image's two-dimensional array of pixels, a single row or "scanline" of the Image, or even a single pixel of data. Multiple calls to setPixels may even define data for overlapping regions of the Image, in which case only the last piece of Image data for a particular region should be used. Images may also contain multiple frames, such as with a video sequence of animation frames.

The ImageProducer calls the ImageConsumer's setHints method to tell the consumer a little about how the Image data will be passed. These hints can be used by a consumer to optimize its processing of the pixel data. Always before the first call to setPixels, setHints is called. A bitwise ORing of ImageConsumer flags is passed to setHints. See the API summary of the setHints method later in this chapter for a comprehensive listing of the ImageConsumer flags.

Producer hints should be used by a consumer to optimize its processing of image data. For example, one custom ImageConsumer might be an average color calculator which calculates the average red, green, and blue intensities of an image. For non-SINGLEPASS images, such a consumer would have to keep a history of data sufficient to subtract overlapping regions passed to setPixels.

Several other important pieces of information about an image are presented to a consumer before individual pixel data is sent to setPixels. The ImageConsumer methods used to pass this additional information are

- setDimensions is called to inform the consumer of the Image's width and height.
- setProperties is called to pass the consumer an extensible list of Image properties. These properties are all stored in a Hashtable as text. The individual properties might include the name and configuration of any filters used to produce the Image, any copyright on the Image, etc.
- setColorModel is passed a copy of the ColorModel used predominantly by the producer. Similar to the hints passed to setHints, the ColorModel may be used by the consumer to optimize processing of Image color data.

When the producer has finished sending the consumer all information about the image (or a single frame of the image), the imageComplete method is invoked. Similar to setHints, imageComplete is passed a bitwise ORing of ImageConsumer flags indicating the status of the completed image. Here are the ImageConsumer flags that may be passed to imageComplete:

| Flag | Meaning |
|---|---|
| IMAGEERROR | An error was encountered while producing the Image data. The error is unrecoverable, and the image data received so far is not necessarily viable. |
| SINGLEFRAMEDONE | A single frame in a multiframe Image has been completed. |
| STATICIMAGEDONE | The Image data for a complete single-frame image has been sent to the consumer. For multi-frame images, this flag implies all frames have been delivered to the consumer. |
| IMAGEABORTED | The production of the Image has been deliberately aborted. |

As you can see, the interpretation of most of these flags is somewhat vague. For example, the IMAGEERROR flag simply indicates that "some error" has occurred: There is no way to discover more about what that error is.

The CountFramesConsumer (see the Example section of the ImageConsumer API Summary below) counts the number of frames in a multiframe Image. The individual pixel data for the frames is actually ignored by the methods in this class. About the only information CountFramesConsumer is interested in is the SINGLEFRAME flag to setHints and the SINGLEFRAMEDONE or STATICIMAGEDONE flags to imageComplete. The CountFramesConsumer's constructor is passed an Image object, whose getSource method returns a Producer to pass the Image's pixel data to a consumer. See the API summary of the ImageConsumer interface for an example of an ImageConsumer.

## ImageFilters

An ImageFilter is a special type of ImageConsumer which passes image data on to another ImageConsumer after suitable processing. Figure 8-3 illustrates ImageFilters as intermediate compartments in the flow of image data from its producer to its ultimate consumer. You can see that multiple filters can be chained together.

**Figure 8-3** ImageFilters are consumers able to pass modified pixel data on to other consumers (including other ImageFilters)

The default implementation of ImageFilter is a null filter. That is, it actually does nothing to the image data, but simply passes it through to another consumer. The filter keeps track of the ImageConsumer to which it is passing data. All other image filters are derived from the ImageFilter class, and override its default implementations of the ImageConsumer interface to actually perform some action on Image data.

Filtering an image involves creating a FilteredImageSource. The FilteredImageSource requires two parameters to its constructor: an ImageProducer and an ImageFilter. The FilteredImageSource simply makes the ImageFilter a consumer of the ImageProducer's data, and funnels the output of the ImageFilter to a new Image object. This code snippet shows how to use an ImageFilter to create a filtered version of a particular image.

```
Image img;
ImageFilter filter;

// img and filter are set to refer to an Image and ImageFilter.

Image imgFiltered = Toolkit.getDefaultToolkit().createImage(
        new FilteredImageSource(img.getSource(), filter));
```

The simplest image filters to create are ones which modify each pixel's RGBα data independant of the pixel data for other pixels in the Image. An example of this type of filter would be a TintFilter, which modifies the tint of each pixel in an Image uniformly, much the same as the tint dial on a television set. AWT provides an RGBImageFilter class to make the creation of such filters much easier.

To create an RGBImageFilter, you need only implement the RGBImageFilter.filterRGB method in your derived RGBImageFilter class. The filterRGB method is passed the coordinates of a single pixel and the color data for that pixel (before any filtering) as an RGBα integer. This method returns a replacement RGBα integer for that particular pixel. In all RGBImageFilters, each individual pixel in an image is passed through filterRGB. The following listing is the code for a simple Color2BWFilter. Color2BWFilter converts a color image to a black-and-white image by making the red, green, and blue components of each individual pixel equal, essentially converting the image to grayscale.

```
class Color2BWFilter extends RGBImageFilter {
```

```
    // The constructor sets
    // RGBImageFilter.canFilterIndexColorModel to true so that
    // index color entries are passed to filterRGB().
    public Color2BWFilter() {
        conFilterIndexColorModel = true;
    }

    // RGB-alpha data passed to filterRGB is converted
    // to grayscale by setting each of the color components
    // to be equal to the average of the r,g and b values.
    public int filterRGB(int x, int y, int rgb) {
        int r = (rgb & 0x00ff0000) >> 16;
        int g = (rgb & 0x0000ff00) >> 8;
        int b = rgb & 0x000000ff;

        // grayscale value is sqrt(r^2 + g^2 + b^2) / sqrt(3).
        int gray = (int)(Math.sqrt((double)(r*r + g*g + b*b)) /
                Math.sqrt((double)3));

        return (gray << 16) | (gray << 8) | gray |
                (rgb & 0xff000000);
    }
}
```

Notice that the constructor of the Color2BWFilter sets a canFilterIndexColorModel member to true. That flag tells the RGBImageFilter implementation that if the image being filtered is based on an IndexColorModel, then passed the IndexColorModel's 256 entries through the filter, not the actual pixel data. In the case of the Color2BWFilter, this means the filter will actually change the values associated with each color index to be a grayscale color. The pixel data doesn't change at all. This feature makes RGBImageFilters remarkably efficient at filtering images based on an IndexColorModel, since only 256 values need to be modified no matter how big the input image is (or, at most the number of color entries in the IndexColorModel, which is usually 256).

More complex filtering than RGBImageFiltering, filtering which is not uniform for all pixels in an image, requires an ImageFilter-derived class which overrides the default implementations of ImageFilter's ImageConsumer methods. An example of such a filtering operation is an image convolution, also known as blurring or sharpening an image. In convolution, the resultant color data for a particular pixel is dependent on the color values of the pixels adjacent to it. To perform this type of operation, your ImageFilter would have to remember the pixel data around the border of regions given to setPixels for filtering.

For example, let's say you need the pixel data of nine pixels from the input image (3 x 3 square of pixels) in order to figure out the pixel data of the center pixel in the output image. You would not be able to calculate the output pixel values for pixels within one pixel of the edge of a rectangle sent to setPixels. You would not be able to calculate these pixel values until the adjacent region was sent to setPixels. In this case, you would have to keep track of the pixel values for these border pixels until you get all adjacent regions.

## AWT Image Processing API Summaries

Table 8-1 lists the classes related to AWT Image processing, which are described in this chapter. Table 8-2 lists the methods associated with the AWT Image processing classes and interfaces, including a short description of each method.

**Table 8-1** The AWT Image processing classes and interfaces summarized

| Class | Description |
|---|---|
| ColorModel | Defines a storage schema for pixel color data. Can convert this data to the default RGBα schema. |
| DirectColorModel | An arbitrary, packed bit storage schema for pixel color data. You can create your own instances with different color component bit widths and masks. |
| IndexColorModel | A storage schema in which each pixel's data is actually an index to a lookup table of colors. |
| ImageConsumer | An Interface implemented by any class whose job is to collect Image pixel data for whatever reason. |
| ImageProducer | An Interface implemented by objects that supply Image pixel data to ImageConsumers. |

**Table 8-2** Methods of the Image processing classes and interfaces

| Class | Method | Description |
|---|---|---|
| ColorModel | getRGBDefault | Gets a ColorModel describing the default RGBα bit packing scheme. |
| | getPixelSize | Gets number of bits of pixel data for the ColorModel. |
| | getRed | Gets red component value given a pixel's value. |
| | getGreen | Gets green component value given a pixel's value. |
| | getBlue | Gets blue component value given a pixel's value. |

| | | |
|---|---|---|
| | getAlpha | Gets transparency value given a pixel's value. |
| | getRGB | Gets RGBα version of a pixel's value. |
| DirectColorModel | getRedMask | Gets bitmask of red component data for pixels based on this DirectColorModel. |
| | getGreenMask | Gets bitmask of green component data for pixels based on this DirectColorModel. |
| | getBlueMask | Gets bitmask of blue component data for pixels based on this DirectColorModel. |
| | getAlphaMask | Gets bitmask of transparency component data for pixels based on this DirectColorModel. |
| IndexColorModel | getMapSize | Gets number of indexed colors in the IndexColorModel. |
| | getTransparentPixel | Gets pixel value reserved for full transparency. |
| | getReds | Fills array with indexed red component values. |
| | getGreens | Fills array with indexed green component values. |
| | getBlues | Fills array with indexed blue component values. |
| | getAlphas | Fills array with indexed transparency values. |
| ImageConsumer | setDimensions | Called to indicate the size of the image. |
| | setProperties | Called to pass the consumer the image's properties. |
| | setColorModel | Indicates the ColorModel for the majority of pixel data being passed to the consumer. |
| | setHints | Indicates some information about the order that pixels will be passed to the consumer. |
| | setPixels | Called to give the consumer image pixel values. |
| | imageComplete | Indicates status of the completed image. |

| | | |
|---|---|---|
| ImageProducer | addConsumer | Registers an ImageConsumer with the producer. |
| | isConsumer | Checks whether an ImageConsumer is registered with this producer. |
| | removeConsumer | Unregisters an ImageConsumer with this producer. |
| | startProduction | Kick-starts the image production process. |
| | requestTopDownLeftRightResend | After image production has completed, asks the producer to re-send the data in a single pass, in top-down, left-right order. |

## ColorModel

**Purpose**

Encapsulate methods for interpreting raw pixel data.

**Syntax**

public abstract class ColorModel

**Description**

Encapsulates methods for interpreting raw pixel data. Image data is associated with a particular ColorModel. The ColorModel can then be used to convert the raw pixel data in RGBα data for the pixel. Figure 8-4 shows the hierarchy of the ColorModel class.

**Package**

*java.awt.image*

**Imports**

None.

**Constructors**

public ColorModel(int pixel_bits);

Since this is an abstract class, this constructor is only called from the constructors of derived classes. Specify the number of bits used to store a single pixel's data; this is the same value returned by getPixelSize.

**Parameters**

***int pixel_bits***

This member stores the number of bits required to store a single pixel's data. It is protected, so it is available to derived classes.

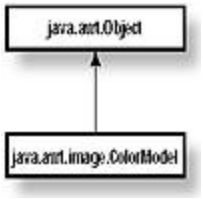


**Figure 8-4** The hierarchy of the ColorModel class

## getRGBDefault

**ClassName**

ColorModel

**Purpose**

Gets a ColorModel object which describes the default RGBα color storage scheme.

**Syntax**

public static ColorModel getRGBDefault();

**Parameters**

None.

**Imports**

None.

**Description**

Returns a DirectColorModel object that defines the default storage schema for RGBα data in an integer. You can use the default RGB ColorModel to unpack red, green, and blue color components from RGBα data. But since the packing schema of the default RGB ColorModel is so well known, it is easier just to unpack the bits yourself. Default RGB packed color data is packed like this: 0xααrrggbb.

**Returns**

The default RGB ColorModel object.

**Example**

The example below actually unpacks default RGB color data using two methods: directly unpacking the data using bitshift operators, and unpacking the data using the default RGB ColorModel:

```
class MyFilter extends RGBImageFilter {

    public int filterRGB(int x, int y, int rgb) {
        // unpack the bits directly…
        int alphaUnpacked = (rgb & 0xff000000) >>> 24;
        int rUnpacked = (rgb & 0x00ff0000) >> 16;
        int gUnpacked = (rgb & 0x0000ff00) >> 8;
        int bUnpacked = rgb & 0x000000ff;

        // use default RGB ColorModel to unpack…
        ColorModel defaultCM = ColorModel.getRGBDefault();
        int alphaFromCM = defaultCM.getAlpha(rgb);
        int rFromCM = defaultCM.getRed(rgb);
        int gFromCM = defaultCM.getGreen(rgb);
        int bFromCM = defaultCM.getBlue(rgb);
```

```
        return rgb;
    }
}
```

## getPixelSize

**ClassName**

ColorModel

**Purpose**

Gets the width of pixel data described by this ColorModel.

**Syntax**

public int getPixelSize();

**Parameters**

None.

**Imports**

None.

**Description**

Gets the number of bits required to store a single pixel's color data. This will be either 8 or 32 bits. IndexColorModels use 8 bits to store the index into a color table. Most other schemas use 32 bits.

**Returns**

The number of bits required to store the color data for a single pixel encoded using this ColorModel.

**Example**

You can use the results of the getPixelSize method to allocate storage for pixel data using a particular ColorModel, as in this example:

```
ColorModel cm;

// Initialize the ColorModel cm…

// Now, allocate an array of 100 elements of the correct
// size to store pixel data.
int[] anPixels = null;
byte[] abPixels = null;

if(cm.getPixelSize() == 32)
    anPixels = new int[100];
else
    abPixels = new byte[100];
```

## getRed

**ClassName**

ColorModel

**Purpose**

Gets value of the red color component for a pixel.

**Syntax**

public int getRed(int pixel);

**Parameters**

***int pixel***

The value of a single pixel's data. If original pixel data is a byte, then it must be cast to an integer.

**Imports**

None.

**Description**

Gets the value of the red color component from an integer of pixel data. This is an abstract method which must be implemented by all ColorModel-derived classes.

**Returns**

The red color component from the pixel data.

**Example**

See the example under the getAlpha method of the ColorModel class, later is this section.

## getGreen

**ClassName**

ColorModel

**Purpose**

Gets value of the green color component for a pixel.

**Syntax**

public int getGreen(int pixel);

**Parameters**

***int pixel***

The value of a single pixel's data. If original pixel data is a byte, then it must be cast to an integer.

**Imports**

None.

**Description**

Gets the value of the green color component from an integer of pixel data. This is an abstract method which must be implemented by all ColorModel-derived classes.

**Returns**

The green color component from the pixel data.

**Example**

See the example under the getAlpha method of the ColorModel class, later in this section.

## getBlue

**ClassName**

ColorModel

**Purpose**

Gets value of the blue color component for a pixel.

**Syntax**

public int getBlue(int pixel);

**Parameters**

***int pixel***

The value of a single pixel's data. If original pixel data is a byte, then it must be cast to an integer.

**Imports**

None.

**Description**

Gets the value of the blue color component from an integer of pixel data. This is an abstract method which must be implemented by all ColorModel-derived classes.

**Returns**

The blue color component from the pixel data.

**Example**

See the example under the getAlpha method of the ColorModel class, later is this section.

## getAlpha

**ClassName**

ColorModel

**Purpose**

Gets the transparency value from a single single's data.

**Syntax**

public int getAlpha(int pixel);

**Parameters**

***int pixel***

The value of a single pixel's data. If original pixel data is a byte, then it must be cast to an integer.

**Imports**

None.

**Description**

Gets the value of the alpha transparency value from an integer of pixel data. Transparency is generally measured from 0-255, where 255 is fully opaque, and 0 is fully transparent. Note also that the IndexColorModel allows you to specify a single index as being fully transparent, similar to the GIF format's transparent color indicator. getAlpha is an abstract method which must be implemented by all ColorModel-derived classes.

**Returns**

The alpha transparency value from the pixel data.

**Example**

In this example, the ColorModel passed to an ImageFilter's setPixels method is used to get the red, green, and blue color components from individual pixels.

```
public MyFilter extends ImageFilter {
    // Overloaded byte version of setPixels() must also be
    // implemented to make a real ImageFilter.
```

```
    public void setPixels(int x, int y, int width, int height
            ColorModel model, int[] pixels, int off,
            int scansz) {
        for( int i=0 ; i<width ; i++ ) {
            for( int j=0 ; j<height ; j++ ) {
                int r=model.getRed(pixels[(j*scansz)+i+off]);
                int g=model.getGreen(pixels[(j*scansz)+i+off]);
                int b=model.getBlue(pixels[(j*scansz)+i+off]);
                int a=model.getAlpha(pixels[(j*scansz)+i+off]);

                // Do something with r, g, b and a values
            }
        }
    }
}
```

## getRGB

**ClassName**

ColorModel

**Purpose**

Converts pixel data to RGBα schema.

**Syntax**

public int getRGB(int pixel);

**Parameters**

***int pixel***

Raw pixel data stored in the ColorModel's storage schema.

**Description**

Converts raw pixel color data to the default RGBα storage schema. The output pixel data is equivalent in color and transparency to the raw color data.

**Returns**

A packed integer of red, green, blue, and alpha color values packed using the default RGB storage schema.

**Example**

This example demonstrates the technique used by the RGBImageFilter to pass all pixel data through the filterRGB method. That is, all pixels passed to RGBImageFilters are converted to the default RGBα storage schema using getRGB:

```
class MyRGBConverter implements ImageConsumer {

    // Other ImageConsumer methods are not implemented here,
    // only one of the overloaded versions of setPixels. In
    // setPixels, the individual pixels are each converted to
    // the default RGB-alpha storage schema before being
    // processed.
    public void setPixels(int x, int y, int width, int height,
            ColorModel model, int[] pixels, int off,
            int scansize) {
        for( int i=0 ; i<width ; i++ ) {
            for( int j=0 ; j<height ; j++ ) {
                int pixel = pixels[(j*scansize)+i+off];
```

```
                processRGB(x, y, model.getRGB(pixel));
            }
        }
    }

    // processRGB does something with the default RGB-alpha
    // data for each pixel. processRGB is very similar to
    // RGBImageFilter.filterRGB.
    public void processRGB(int x, int y, int rgb) {
        // Do something with RGB data here.
    }
}
```

## DirectColorModel

**Purpose**

Encapsulates methods for interpreting image pixel data based on packed bitfields of red, green, blue, and alpha color components.

**Syntax**

public class DirectColorModel extends ColorModel

**Description**

The DirectColorModel encapsulates methods for interpreting image pixel data based on packed bitfields of red, green, blue, and alpha color components. The size and presence of the bitfields for each of these components is completely configurable. You could, for example, create a DirectColorModel object which can interpret 5-6-5 16-bit RGB pixel data (16 bits of each pixel's data are used, top 5 bits are red, next 6 are green, and bottom 5 are blue). The example below demonstrates how to create such a DirectColorModel. Figure 8-5 shows the hierarchy of the DirectColorModel class.

**Package**

*java.awt.image*

**Imports**

*java.awt.AWTException*

**Constructors**

public DirectColorModel(int *pixel_bits*, int *redmask*, int *greenmask*, int *bluemask*);

public DirectColorModel(int *pixel_bits*, int *redmask*, int *greenmask*, int *bluemask*, int *alphamask*);

The *pixel_bits* parameter can be 8 or 32 bit, and it specifies how many bits of pixel data are used per pixel. Specify bitmasks for each of the color components in the mask parameters. If it is left out, the *alphamask* is assumed to be 0x00000000.

**Parameters**

None.

**Example**

This example builds a DirectColorModel that interprets color data using the default RGBα storage schema: 0xααrrggbb.

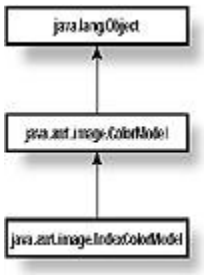


**Figure 8-5** The hierarchy of the DirectColorModel class

```
//…

DirectColorModel dcm =
    new DirectColorModel(32,  // 32 bits/pixel
                         0x00ff0000, // the red mask
                         0x0000ff00, // the green mask
                         0x000000ff, // the blue mask
                         0xff000000); // the alpha mask

//…
```

This example creates a DirectColorModel that interprets color data using a 5-6-5 16-bit RGB shema:

```
DirectColorModel dcm =
    new DirectColorModel(32,  // 32 bits/pixel
                         0x0000F800, // red mask
                         0x000007E0, // green mask
                         0x0000001F, // green mask
                         0x00000000); .// no alpha data
```

## getRedMask

**ClassName**
DirectColorModel
**Purpose**
Gets bitmask of packed red component values.
**Syntax**
public int getRedMask();
**Parameters**
None.
**Imports**
None.
**Description**
Gets the bitmask defining the bitfield in which red color data is stored in packed integers using this DirectColorModel.
**Returns**
The red bitmask passed to the DirectColorModel's constructor.

**Example**

See the example under the getAlphaMask method of the DirectColorModel, later in this section.

## getGreenMask

**ClassName**

DirectColorModel

**Purpose**

Gets bitmask of packed green component values.

**Syntax**

public int getGreenMask();

**Parameters**

None.

**Imports**

None.

**Description**

Gets the bitmask defining the bitfield in which green color data is stored in packed integers using this DirectColorModel.

**Returns**

The green bitmask passed to the DirectColorModel's constructor.

**Example**

See the example under the getAlphaMask method of the DirectColorModel, later in this section.

## getBlueMask

**ClassName**

DirectColorModel

**Purpose**

Gets bitmask of packed blue component values.

**Syntax**

public int getBlueMask();

**Parameters**

None.

**Imports**

None.

**Description**

Gets the bitmask defining the bitfield in which blue color data is stored in packed integers using this DirectColorModel.

**Returns**

The blue bitmask passed to the DirectColorModel's constructor.

**Example**

See the example under the getAlphaMask method of the DirectColorModel, later in this section.

## getAlphaMask

**ClassName**

DirectColorModel

**Purpose**

Gets bitmask of packed alpha or transparency component values.

**Syntax**

public int getAlphaMask();

**Parameters**

None.

**Imports**

None.

**Description**

Gets the bitmask defining the bitfield in which alpha color data is stored in packed integers using this DirectColorModel.

**Returns**

The alpha bitmask passed to the DirectColorModel's constructor.

**Example**

Given an Image which uses a DirectColorModel, this example method displays how many bits are used to store each color component in the packed raw image data.

```
public void displayBitSizes(Image img) {
    // Make sure Image uses DirectColorModel
    if(! (img.getColorModel() instanceof DirectColorModel))
        return;

    DirectColorModel dcm = img.getColorModel();
    int redmask = dcm.getRedMask();
    int greenmask = dcm.getGreenMask();
    int bluemask = dcm.getBlueMask();
    int alphamask = dcm.getAlphaMask();

    int redbits, greenbits, bluebits, alphabits;
    for(int ii=0 ; ii<32 ; ii++)
        redbits += (0 != redmask & 0x00000001 ? 1 : 0);
        greenbits += (0 != greenmask & 0x00000001 ? 1 : 0);
        bluebits += (0 != bluemask & 0x00000001 ? 1 : 0);
        alphabits += (0 != alphamask & 0x00000001 ? 1 : 0);

        redmask >>= 1;
        greenmask >>= 1;
        bluemask >>= 1;
        alphamask >>= 1;
    }

    System.out.println("red field width is " + redbits);
    System.out.println("green field width is " + greenbits);
    System.out.println("blue field width is " + bluebits);
    System.out.println("alpha field width is " + alphabits);
}
```

## IndexColorModel

**Purpose**

Encapsulates methods for interpreting image pixel data based on an indexed color palette. Most IndexColorModels describe a 256-color palette, although the IndexColorModel class constructor allows you to specify any number up to 4 billion.

**Syntax**

public class IndexColorModel extends ColorModel

**Description**

For some images, the image pixel data is really an index into a color table. The IndexColorModel represents the color table. The class provides methods for converting pixel data into actual red, green, blue, and alpha color component values. Figure 8-6 shows the hierarchy of the IndexColorModel class.

**Package**

*java.awt.image*

**Imports**

None.

**Constructors**

public IndexColorModel(int *bits*, int *size*, byte[] *r*, byte[] *g*, byte[] *b);*
public IndexColorModel(int *bits*, int *size*, byte[] *r*, byte[] *g*, byte[] *b*, int *trans*);
public IndexColorModel(int *bits*, int *size*, byte[] *r*, byte[] *g*, byte[] *b*, byte[] *a*);
public IndexColorModel(int *bits*, int *size*, byte[] *cmap*, int *offset*, boolean *hasalpha*);
public IndexColorModel(int *bits*, int *size*, byte[] *cmap*, int *offset*, boolean *hasalpha*, int *trans*);

The *bits* parameter specifies the number of bits of pixel data associated with each pixel. The *size* parameter indicates the size of the color map, and the color map is described by the *r*, *g*, *b*, and *a* arrays, or by the *cmap* array.

The *cmap* array is assumed to have the pattern [r|g|b|a|r|g|b|a|…] or [r|g|b|r|g|b|…]. That is, every third or fourth byte specifies the red color component value for index entry (byte number/4). Every third or fourth starting at index 1 encodes the green color component value for successive index entries. Every third or fourth starting at index 2 encodes the blue component value for successive entries, and so on.

If the *hasalpha* parameter is true, then every fourth *cmap* byte, starting at index 3, encodes the alpha values for successive index entries.

The *trans* parameter indicates the index which is to be interpreted as "transparent." For example, GIF images allow you to specify a "transparent color." All pixels of that color are rendered as transparent when the image is drawn.

**Parameters**

None.

**Example**

This example builds a 256-entry IndexColorModel with colors distributed evenly around the color wheel. A black color is defined, a white color and a transparent

color, which leaves room for 253 more colors. Even distribution using 6 increments of red, 6 increments or blue, and 7 increments of green creates 252 more color entries. That leaves a single entry which will be defined as gray:

```
//…

byte r[256];
byte g[256];
byte b[256];

// Create the white, black, gray, and transparent entries as the
// last four entries in each array.
r[255] = g[255] = b[255] = 0; // black
r[254] = g[254] = b[254] = 255; // white
r[253] = g[253] = b[253] = 128; // gray
r[252] = g[252] = b[252] = 0; // used for transparency

// Create evenly distributed sets of RGB entries by allowing
// 6 even increments of red, 6 even increments of blue, and
// 7 even increments of green = 6*6*7 = 252 entries.
for( int iR=0 ; iR<6 ; iR++ ) {
    for( int iB=0 ; iB<6 ; iB++ ) {
        for( int iG=0 ; iG<7 ; iG++ ) {
            int index = iR + (6*iB) + (36*iG);
            r[index] = (iR*42) + 21;
            b[index] = (iB*42) + 21;
            g[index] = (iG*36) + 18;
        }
    }
}

// Create the IndexColorModel
IndexColorModel icm =
        new IndexColorModel(8, // pixels are 8 bit each
                            256, // size of color arrays
                            r, // the red entries
                            g, // the green entries
                            b, .. the blue entries
                            252); // transparent index

//…
```

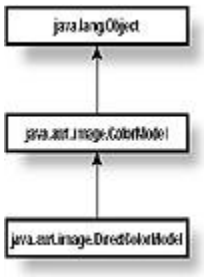


**Figure 8-6** The hierarchy of the IndexColorModel class

## getMapSize

**ClassName**

IndexColorModel

**Purpose**

Gets the number of color entries in the IndexColorModel.

**Syntax**

public int getMapSize();

**Parameters**

None.

**Imports**

None.

**Description**

Gets the number of color entries in the IndexColorModel. This is the same value as the size parameter to the IndexColorModel's constructor. The arrays you pass to the getReds, getGreens, getBlues, or getAlphas methods must be at least this long.

**Returns**

Returns the number of color entries in the IndexColorModel.

**Example**

This code creates a two-dimensional array to hold the indexed color entries.

```
public byte[][] getRGBAs(IndexColorModel icm) {
    byte[][] aabRet = new byte[4][icm.getMapSize()];
    icm.getReds(aabRet[0]);
    icm.getGreens(aabRet[1]);
    icm getBlues(aabRet[2]);
    icm getAlphas(aabRet[3]);
    return aabRet;
}
```

## getTransparentPixel

**Class**

IndexColorModel

**Purpose**

Gets the index of the "transparent color" of the IndexColorModel.

**Syntax**

public int getTransparentPixel();

**Parameters**

None.

**Imports**

None.

**Description**

Gets the index of the "transparent color" of the IndexColorModel. This is the same as the *trans* parameter to the IndexColorModel constructor.

**Returns**

The value of the "transparent index" is returned. If there is no transparent color, then -1 is returned.

**Example**

In this example ImageFilter, that assumes the ColorModel is an IndexColorModel, all pixels of a particular color are converted to the transparent color.

```
public class TransColorFilter extends ImageFilter {
    byte _substColor;

    public TransColorFilter(byte color_index) {
        _substColor = color_index;
    }

    public void setPixels(int x, int y, int w, int h,
            ColorModel model, byte[] pixels, int off,
            int scansize) {
        if!(model instanceof IndexColorModel)) {
            consumer.setPixels(x, y, w, h, model, pixels,
                    off, scansize);
            return;
        }

        IndexColorModel icm = (IndexColorModel)model;
        int trans = icm.getTransparentPixel();
        if(-1 == trans) {
            consumer.setPixels(x, y, w, h, model, pixels,
                    off, scansize);
            return;
        }

        for(int xx=x ; xx<x+w ; xx++)
            for(int yy=y ; yy<y+h ; yy++)
                if(pixels[xx * scansize + yy + off] ==
                        substColor)
                    pixels[xx * scansize + yy + off] =
                            (byte)trans;
        consumer.setPixels(x, y, w, h, model, pixels,
                off, scansize);
    }

…
}
```

## getReds

**ClassName**

IndexColorModel

**Purpose**

Gets index of red color component values.

**Syntax**

public void getReds(byte[] r);

**Parameters**

***byte[] r***

Array of bytes. This array must have as many elements as indicated by the getMapSize method.

**Imports**

None.

**Description**

Gets an array of red color components. The values of the internal array of red color values is copied to the passed byte array.

**Returns**

None.

**Example**

See the example for the getAlphas method.

## getGreens

**ClassName**

IndexColorModel

**Purpose**

Gets index of green color component values.

**Syntax**

public void getGreens(byte[] g);

**Parameters**

***byte[] g***

Array of bytes. This array must have as many elements as indicated by the getMapSize method.

**Imports**

None.

**Description**

Gets an array of green color components. The values of the internal array of green color values is copied to the passed byte array.

**Returns**

None.

**Example**

See the example for the getAlphas method.

## getBlues

**ClassName**

IndexColorModel

**Purpose**

Gets index of blue color component values.

**Syntax**

public void getBlues(byte[] b);

**Parameters**

***byte[] b***

Array of bytes. This array must have as many elements as indicated by the getMapSize method.

**Imports**

None.

**Description**

Gets an array of blue color components. The values of the internal array of blue color values is copied to the passed byte array.

**Returns**

None.

**Example**

See the example for the getAlphas method.

## getAlphas

**ClassName**

IndexColorModel

**Purpose**

Gets index of alpha transparency component values.

**Syntax**

public void getAlphas(byte[] a);

**Parameters**

***byte[] a***

Array of bytes. This array must have as many elements as indicated by the getMapSize method.

**Imports**

None.

**Description**

Gets an array of transparency components. The values of the internal array of transparency values is copied to the passed byte array.

**Returns**

None.

**Example**

This example writes any IndexColorModel's full palette of colors to System.out.

```
public void displayPalette(IndexColorModel icm) {
    int nSize = icm.getMapSize();
    byte[] abReds = new byte[nSize];
    byte[] abGreens = new byte[nSize];
    byte[] abBlues = new byte[nSize];
    byte[] abAlphas = new byte[nSize];

    icm.getReds(abReds);
    icm.getGreens(abGreens);
    icm.getBlues(abBlues);
    icm.getAlphas(abAlphas);

    for(int ii=0 ; ii<nSize ; ii++)
        System.out.println(ii + "=" +
                "r:" + abReds[ii] +
                ", g:" + abGreens[ii] +
                ", b: " + abBlues[ii] +
                ", a: " + abAlphas[ii]);
}
```

## ImageConsumer

**Purpose**

Implement the ImageConsumer interface in your object to receive image pixel data from an ImageProducer.

**Syntax**

interface ImageConsumer

**Description**

The ImageConsumer interface is implemented by objects wishing to process image data. The ImageConsumer is associated with an ImageProducer, and the producer sends image pixel data to the consumer through the setPixels method. Figure 8-7 shows the hierarchy of the ImageConsumer interface.

**Package**

*java.awt.image*

**Imports**

None.

**Example**

This ImageConsumer counts the number of frames in an image.

**Figure 8-7** The hierarchy of the ImageConsumer interface

```
class CountFramesConsumer implements ImageConsumer {

    // accumulator to keep count of the number of frames seen.
    int _nFrames = 0;

    // Flag to indicate a frame has had an error or has been
    // aborted, as reported to imageComplete().
    boolean _fErrorEncountered = false;
    boolean _fComplete = false;

    // Users of this class should ask isComplete() to see
    // if counting of frames has completed yet.
    public boolean isComplete() {
        return _fComplete;
    }

    // Get the frame count using getFrameCount().
    public int getFrameCount() {
        return _nFrames;
    }

    // constructor requires an Image to count the frames for.
    public CountFramesConsumer(Image img) {
        ImageProducer p = img.getSource();
        p.addConsumer(this);
        p.startProduction(this);
    }

    // To increment the frame counter, the _fComplete flag
```

```
    // must not be set.
    public void incrementFrameCount() {
        if( false == _fComplete )
            _nFrames++;
    }

    // Ignore the dimensions, though an implementation of this
    // method is required.
    public void setDimensions(int width, int height) {
    }

    // Ignore properties, though implementation of this method
    // is required.
    public void setPropertires(Hashtable props) {
    }

    // Ignore ColorModel, though implementation of this method
    // is required.
    public void setColorModel(ColorModel model) {
    }

    // Hints might be useful, especially if the SINGLEFRAME hint
    // is passed, indicating calculation is complete.
    public void setHints(int flags) {
        if(0 != (flags & SINGLEFRAME)) {
            incrementFrameCount();
            _fComplete = true;
        }
    }

    // Pixel data can actually be ignored, though both
    // overloaded versions of setPixels() must be implemented.
    public void setPixels(int x, int y, int width, int height,
            byte[] pixels, int off, int scansize) {
    }

    public void setPixels(int x, int y, int width, int height,
            int[] pixels, int off, int scansize) {
    }

    // imageComplete is called whenever an image has been
    // completed. Look out for the IMAGEERROR and IMAGEABORTED
    // flags.
    public void imageComplete(int flags) {
        if(0 != (flags & (IMAGEERROR|IMAGEABORTED)))
            _fErrorEncountered = true;

        if(0 != (flags & SINGLEFRAMEDONE))
            incrementFrameCount();

        if(0 != (flags & STATICIMAGEDONE))
            _fComplete = true;
    }
}
```

## setDimensions

**Interface**

ImageConsumer

**Purpose**

Called to indicate the dimensions of the image being processed by the ImageConsumer.

**Syntax**

public void setDimensions(int width, int height);

**Parameters**

***int width***

The width of the Image being presented to the consumer.

***int height***

The height of the Image being presented to the consumer.

**Imports**

None.

**Description**

The consumer is passed the dimensions of the Image that is being passed to it by the consumer through this method. The setDimensions method is called by the producer exactly once. This call is guaranteed to occur before any call to setPixels.

**Example**

This example allocates storage for processing an Image based on the size of the Image. The setColorModel method is also implemented so that the ImageConsumer knows the size of each pixel's data.

```
class MyConsumer implements ImageConsumer {
    int[] anImageProcessingData;
    byte[] abImageProcessingData;
    boolean fIsByteSized = true;
    Dimension dim = null;

    public void setDimensions(int width, int height) {
        dim = new Dimension(width, height);
        aloocate();
    }

    public void setColorModel(ColorModel cm) {
        bIsByteSized = (8 == cm.getPixelSize());
        allocate();
    }

    public void allocate() {
        if(null == dim)
            return;

        if(fIsByteSized) {
            abImageProcessingData =
                    new byte[dim.width * dim.height];
            anImageProcesingData = null;
        } else {
            anImageProcessingData =
                    new int[dim.width * dim.height];
            abImageProcessingData = null;
        }
```

```
    }

    // Other consumer methods implemented…
}
```

## setProperties

**Interface**

ImageConsumer

**Purpose**

Called to give the ImageConsumer a list of the image's properties.

**Syntax**

public void setProperties(Hashtable props);

**Parameters**

***Hashtable props***

Image properties are stored under String keys in the props Hashtable. Properties may include the filters used to create the Image data, the source of the Image data, etc.

**Returns**

None.

**Description**

An extensible list of Image properties is passed to the consumer by the producer. The properties are stored in a Hashtable object. Currently, there are only two documented image properties: comments and filter. The comments property has a textual description of the image. The filter property describes which filters were applied to the original Image to get the pixel data being received by this ImageConsumer. setProperties is guaranteed to be called before any calls to setPixels.

**Example**

This example displays all the Image properties on System.out.

```
class MyConsumer extends ImageConsumer {

    public void setProperties(Hashtable props) {
        System.out.println( "MyConsumer properties: " + props);
    }

    // Other consumer methods implemented…
}
```

## setColorModel

**Interface**

ImageConsumer

**Purpose**

Sets the ColorModel for the majority of the image pixel data.

**Syntax**

public void setColorModel(ColorModel model);

**Parameters**

***ColorModel model***

The ColorModel, which will be used in the majority of calls to setPixels().

**Imports**

None.

**Description**

The producer passes to the consumer a ColorModel that will be used in the majority of calls to setPixels. No assumption should be made by the ImageConsumer that model will be the only ColorModel passed to the consumer's setPixels method, since each set of pixels can be sent with a unique ColorModel object. For example, an ImageFilter may pass a majority of pixels on to the consumer without modification. For a minority of those pixels, the filter may use a ColorModel conducive to the ImageFilter's filtering process. In that case, the original producer's ColorModel will be passed to the consumer through the filter in the majority of calls to setPixels, and it will be passed to setColorModel. Note also that setColorModel will be called before the first call to setPixels.

**Returns**

None.

**Example**

See the example under the setDimensions method of the ImageConsumer interface.

## setHints

**Interface**

ImageConsumer

**Purpose**

Provides the consumer with some practical information about how pixel data will be sent to it.

**Syntax**

public void setHints(int flags);

**Parameters**

***int flags***

A set of ImageConsumer flags OR together bitwise. The valid flags, and a description of each, is presented in the following table:

| Flag | Meaning |
|---|---|
| RANDOMPIXELORDER | No assumption can be made about the order in which pixels will be delivered to the consumer. |
| TOPDOWNLEFTRIGHT | Pixels will be delivered to the consumer in top-down, left-right order. |
| COMPLETESCANLINES | Pixels will be delivered to the consumer in multiples of complete scanlines. That is, all calls to setPixels() will |

| | |
|---|---|
| | have a width parameter equal to the width of the Image. |
| SINGLEPASS | Each pixel's data will appear in exactly one call to setPixels(). |
| SINGLEFRAME | The Image includes only a single frame. A JPEG image, for example, includes only a single frame. An MPG Image, on the other hand, includes multiple frames of an animation sequence. |

---

**Imports**

None.

**Description**

The producer can pass to the consumer some hints about the order that pixels will be passed to the consumer through setPixels. Those hints are flags passed to the setHints method.

The hints passed to the consumer in setHints can be used by the consumer to optimize its processing of image data. For example, a Fourier Transform consumer could realize significant optimizations in its storage and processing using the so-called "fast-Fourier transform" algorithm if the Image data is passed as complete scanlines, indicated by the COMPLETESCANLINES flag being passed to this method.

**Returns**

None.

**Example**

This ImageConsumer calculates the average red, green, and blue color component values for all the pixels it is passed. The consumer is pretty simple, however, and requires that each pixel's value be sent to it only once. This consumer waits for the SINGLEPASS flag to be passed to its setHints method. If this flag is not passed, then the consumer doesn't even try to accomplish its task, since it would be too hard to keep track of the number of times each pixel was re-sent to the consumer.

```
public class MyConsumer implements ImageConsumer {
    int _sumRed, _sumGreen, _sumBlue;
    int _nPixels;
    boolean _fComplete;
    boolean _fCalc;

…

    public MyConsumer() {
        _sumRed = _sumGreen = _sumBlue = 0;
        _nPixels = 0;
        _fComplete = false;
    }

    public void setHints(int flags) {
        _fCalc = (0 != (flags & SINGLEPASS));

    }
```

…

```
    public void setPixels(int x, int y, int w, int h,
            ColorModel model, byte[] pixels, int offset,
            int scansize) {
        if(!_fCalc) return;

        for(int xx=x ; xx<x+w ; xx++)
            for(int yy=y ; yy<y+h ; yy++) {
                byte p = pixels[xx*scansize+yy+offset];
                _sumRed += model.getRed(p);
                _sumGreen += model.getGreen(p);
                _sumBlue += model.getBlue(p);
                _nPixels++;
            }
    }

    public void imageComplete(int status) {
        // status parameter not used.
        _fComplete = true;

    // Use these functions to get the results
    // of the averaging.
    public float getAverageRed() {
        if(!_fCalc || 0==_nPixels) return (float)0;
        return (float)_sumRed / (float)_nPixels;
    }

    public float getAverageGreen() {
        if(!_fCalc || 0==_nPixels) return (float)0;
        return (float)_sumGreen / (float)_nPixels;
    }

    public float getAverageBlue() {
        if(!_fCalc || 0==_nPixels) return (float)0;
        return (float)_sumBlue / (float)_nPixels;
    }

    // This method returns true when the consumer
    // is finished with its processing.
    public boolean isComplete() {
        return _fComplete;
    }
}
```

## setPixels

**Interface**

ImageConsumer

**Purpose**

All pixel data sent to a consumer is sent through one or both of the overloaded setPixels methods.

**Syntax**

public void setPixels(int x, int y, int width, int height, ColorModel model, byte[] pixels, int offset, int scansize);public void setPixels(int x, int y, int width, int height, ColorModel model, int[] pixels, int offset, int scansize);

**Parameters**

***int x, int y***

The origin or upper-left corner of the rectangle of pixels being passed to the consumer.

***int width, int height***

The dimensions of the rectangle of pixels being passed to the consumer.

***ColorModel model***

The ColorModel defining the storage schema of color and transparency data for the *pixels[]* array.

***byte pixels[]***

An array of 8-bit pixel data. The *x, y, width, height, offset* and *scansize* parameters must be used to determine which elements of this array contain valid pixel data for the Image.

***int pixels[]***

An array of 32-bit pixel data. The *x, y, width, height, offset* and *scansize* parameters must be used to determine which elements of this array contain valid pixel data for the Image.

***int offset***

Used along with *scansize* to determine the valid elements of the *pixels[]* array.

***int scansize***

Used along with *offset* to determine the valid elements of the *pixels[]* array.

**Imports**

*java.awt.image.ColorModel*

**Description**

Image pixels are passed from the producer to the consumer in multiple calls to setPixels. Each call defines the pixel data for a rectangle of the image. Two overloaded versions of this method exist, one to accept 8-bit pixel data, and the other to accept 32-bit pixel data.

Use this formula to find the pixels array element for pixel (M, N):

```
p(M,N) = pixels[(n * scansize) + M + offset];
```

**Returns**

None.

**Example**

The following example, which assumes the SINGLEPASS flag has been passed to setHints, calculates the average pixel values of red, green, and blue for all pixel data passed to MyConsumer:

```
class MyConsumer implements ImageConsumer {
    double _dRedAccum = 0;
    double _dGreenAccum = 0;
    double _dBlueAccum = 0;
    double _dTotalPixels = 0;
    int _nAverageRed = 0;
    int _nAverageGreen = 0;
    int _nAverageBlue = 0;
```

```
// other consumer methods implemented…

    public void setDimensions(int width, int height) {
        _dTotalPixels = (double)width * (double)height;
    }

    // For the sake of brevity, only one of the overloaded
    // versions of setPixels() is implemented here, although
    // your ImageConsumer must implement both the 8-bit and
    // 32-bit versions.
    public void setPixels(int x, int y, int width, int height,
            ColorModel model, int[] pixels, int offset,
            int scansize) {
        for( int I=0 ; i<width ; i++ ) {
            for( int j=0 ; j<height ; j++ ) {
                int pixel = pixels[(j*scansize)+i+offset];
                _dRedAccum += (double)model.getRed(pixel);
                _dGreenAccum += (double)model.getGreen(pixel);
                _dBlueAccum += (double)model.getBlue(pixel);
            }
        }
    }

    public void imageComplete(int flags) {
        if(0 != (flags & STATICIMAGEDONE)) {
            _nAverageRed = (int)(_dRedAccum/_dTotalPixels);
            _nAverageGreen = (int)(_dGreenAccum/_dTotalPixels);
            _nAverageBlue = (int)(_dBlueAccum/_dTotalPixels);
        }
    }
}
```

## imageComplete

**Interface**

ImageConsumer

**Purpose**

Called to notify the consumer that all image pixel data has been sent.

**Syntax**

public void imageComplete(int flags);

**Parameters**

***int flags***

A bitwise ORing of ImageConsumer flags indicating the state of the production sequence upon termination. The following table lists the ImageConsumer flags which can be passed to imageComplete.

| Flag | Meaning |
|---|---|
| IMAGEERROR | The producer encountered an error while processing the Image data. No more image data will be passed to the consumer for this Image. |

| | |
|---|---|
| SINGLEFRAMEDONE | A single frame in a multiframe Image has been completed. |
| STATICIMAGEDONE | The entire image, whether a single-frame or a multiframe Image, has been completely sent to the consumer. |
| IMAGEABORTED | The Image creation process was deliberately aborted. |

---

**Imports**

None.

**Description**

This method is called by the producer to notify the consumer that all image pixel data has been sent to the consumer. The reason for the completion can be an error, a single-frame of a multiframe Image has been completed, or the entire Image has been completed. The flags passed to this method indicate the nature of the completion.

Unfortunately, more specific information about the interpretation of IMAGEERROR or IMAGEABORTED flags is not available to the consumer, unless you implement custom methods in your consumers and producers to provide this information.

**Example**

See the Example for the setPixels method.

## ImageProducer

**Purpose**

An ImageProducer generates image pixel data and sends it to one or more ImageConsumers.

**Syntax**

interface ImageProducer

**Description**

To have your object generate images for display, have it implement the ImageProducer interface. The interface sends image pixel data based on a ColorModel. The image pixel data is sent in potentially overlapping rectangles of image data. Figure 8-8 shows the hierarchy of the ImageProducer interface.

**Package**

java.awt.image

**Imports**

None.

**Example**

This example ImageProducer generates a color gradient. Specify to the constructor the size of the image you want produced and the colors at the top and bottom of the image.

**Figure 8-8** The hierarchy of the ImageProducer interface

```
public class GradientProducer implements ImageProducer {
    private Dimension dim;
    private float flRedInc, flGreenInc, flBlueInc;
    private Color colorStart;
    private Vector vectConsumers = new Vector();

    public GradientProducer(Dimension size, Color top,
            Color bottom) {
        dim = new Dimension(size.width, size.height);
        flRedInc = (float)(bottom.getRed()-top.getRed()) /
                    (float)dim.height;
        flGreenInc = (float)(bottom.getGreen()-
                             top.getGreen()) /
                    (float)dim.height;
        flBlueInc = (float)(bottom.getBlue()-
                            top.getBlue()) /
                    (float)dim.height;
        colorStart = top;
    }

    public void addConsumer(ImageConsumer ic) {
        if(isConsumer(ic))
            return;
        vectConsumer.append((Object)ic);
    }

    public boolean isConsumer(ImageConsumer ic) {
        return vectConsumers.contains((Object)ic);
    }

    public void removeConsumer(ImageConsumer ic) {
        vectConsumer.remove((Object)ic);
    }

    // To produce the image, create each scanline
    // of the next color in the gradient and give
    // it to the ImageConsumer through setPixels.
    // We are going to use default RGBα encoding.
    public void startProduction(ImageConsumer ic) {
        if(!isConsumer(ic))
            return;

        ic.setHints(ImageConsumer.COMPLETE_SCANLINES |
                    ImageConsumer.SINGLE_FRAME);

        int[] apixels = new int[dim.width];
        for(int ii=0 ; ii<dim.height ; ii++) {
            Color c = new Color(colorStart.getRed() +
                                (int)(flRedInc * ii),
                                colorStart.getGreen() +
                                (int)(flGreenInc * ii),
                                colorStart.getBlue() +
                                (int(flBlue * ii));
            int rgb = c.getRGB();
            for(int jj=0 ; jj<dim.width ; jj++)
                apixels[jj] = rgb;
            ic.setPixels(0, ii, dim.width, 1,
```

```
                    ColorModel.getRGBDefault(), apixels,
                    -ii * dim.width, dim.width);
        }

        ic.imageComplete(ImageConsumer.STATICIMAGEDONE);
    }

    // Ignore requests for re-sends.
    public void requestTopDownLeftRightResend() {
        return;
    }
}
```

## addConsumer

**Interface**

ImageProducer

**Purpose**

Registers an ImageConsumer with the ImageProducer. Only registered consumers can receive pixel data.

**Syntax**

public void addConsumer(ImageConsumer ic)

**Parameters**

***ImageConsumer ic***

A new ImageConsumer that the producer is to send pixel data to. Note that the producer should take care not to allow the same consumer to be added multiple times.

**Imports**

None.

**Description**

The producer is asked to send pixel data to a new ImageConsumer. The producer must keep track of more than one consumer, making sure to send each of its consumers all pixel data for an Image.

The producer may wait for an explicit call to startProduction, or it may begin sending pixel data to the consumer as soon as it is ready.

**Returns**

None.

**Example**

This code stores the list of currently active consumers in a Vector. This code is suitable for use in your own ImageProducers to keep track of ImageConsumer objects.

```
class MyProducer implements ImageProducer {
    Vector _vectConsumers;

    public void addConsumer(ImageConsumer ic) {
        if(isConsumer((Object)ic))
            return;

        _vectConsumer.add((Object)ic);
    }
```

```
    public boolean isConsumer(ImageConsumer ic) {
        return _vectConsumers.contains((Object)ic);
    }

    public void RemoveConsumer(ImageConsumer ic) {
        _vectConsumers.remove((Object)ic);
    }

    // startProduction and requestTopDownLeftRightResend
    // must be implemented to create a complete producer.
}
```

## isConsumer

**Interface**
ImageProducer
**Purpose**
Checks to see if an ImageConsumer is registered with this ImageProducer.
**Syntax**
public boolean isConsumer(ImageConsumer ic)
**Parameters**
***ImageConsumer ic***
The ImageConsumer to check.
**Imports**
None.
**Description**
Tells whether or not an ImageConsumer is currently registered with this producer.
**Returns**
True if the ImageConsumer is currently registered with the ImageProducer.
Otherwise, false.
**Example**
See the example listed for addConsumer.

## removeConsumer

**Interface**
ImageProducer
**Syntax**
public void removeConsumer(ImageConsumer ic)
**Parameters**
***ImageConsumer ic***
The ImageConsumer to remove as a consumer of this producer's pixel data.
**Imports**
None.
**Description**

Removes the ImageConsumer as a consumer of pixel data from this producer. If the consumer is not currently registered with this producer, this method should return benignly.

**Example**

See the example listed for addConsumer.

## startProduction

**Interface**

ImageProducer

**Purpose**

Tells the ImageProducer to start sending pixel data to a specific registered ImageConsumer.

**Syntax**

public void startProduction(ImageConsumer ic)

**Parameters**

***ImageConsumer ic***

The consumer to begin sending pixel data to.

**Imports**

None.

**Description**

This method can be called by any external object to kick-start an image production process. An ImageProducer may start sending image pixel data to an ImageConsumer as soon as the consumer gets registered with the producer using addConsumer. The ImageProducer is not required to send any pixel data to a consumer until startProduction is called.

If the consumer has not been added to this producer yet, then this method should return benignly. The minimum implementation of startProduction should call these ImageConsumer methods:

```
ic.setHints(<hints>);
ic.setPixels(…); // as many times as needed.
ic.imageComplete(STATICIMAGEDONE);
```

**Returns**

None.

**Example**

See the example under the description of the ImageProducer interface.

## requestTopDownLeftRightResend

**Interface**

ImageProducer

**Purpose**

To force the producer to re-send all image data in a single pass, in top-down-left-right order.

**Syntax**

public void requestTopDownLeftRightResend(ImageConsumer ic)

**Parameters**

***ImageConsumer ic***

The consumer to re-send the Image data to. If the consumer has not been added to this producer yet, then this method should return benignly.

**Imports**

None.

**Description**

A request to re-send Image pixel data in TOPDOWNLEFTRIGHT order. The producer can choose whether or not to grant or ignore this request.
Producers, which present Image data from sources which are too diffucult to deliver in TOPDOWNLEFTRIGHT order, need not grant this re-send request.

**Returns**

None.

## The AWT Image Processing Project: The MultiFilter Application

The MultiFilter application is a generic application for testing ImageFilters. It is able to load and display Images from the Internet, or the local file system. MultiFilter's user-interaction involves selecting filters to apply to an image as a whole, or to a selected rectangle of the image. MultiFilter is extensible, so you can add your own ImageFilters to it easily. Once you understand the workings of MultiFilter, you are encouraged to try adding some of the sample filters presented earlier in this chapter, or filters of your own.

Two new ImageFilters are included in the MultiFilter application: The ContrastFilter demonstrates how to build RGBImageFilters. The InvertFilter is a more complex image filter, capable of flipping an Image horizontally or vertically, and is thus directly derived from ImageFilter.

In addition to the AWT Image Processing techniques demonstrated by MultiFilter, this project also demonstrates a user-interface model using modeless dialogs to configure the individual ImageFilters. In this model, user interactions with floating modeless dialogs are reported to the application's main Frame window using the Event delivery pipeline inherited by all Component objects.

Figure 8-9 is a screenshot of the MultiFilter application running. You can see the image that is loaded has had both the contrast and invert filters applied to it. Figures 8-10 and 8-11 are before and after pictures of this image, illustrating how image filtering can be applied to an image.

**Figure 8-9** Screenshot of the MultiFilter application running in Windows

**Figure 8-10** Before an image is processed by MultiFilter

**Figure 8-11** After an image is processed by MultiFilter

### Assembling the Project

Note that all the files listed here are also available on the CD packaged with this book in the directory "\PROJECTS\GO\HERE\MultiFilter". Please feel free to copy those files directly.

**1.** Create a new directory called MultiFilter.

**2.** Create a new file in the MultiFilter directory called "MultiFilter.java". Start by importing the necessary packages, declaring our application object, and implementing the static main() method.

```
import java.awt.*;
import java.awt.image.*;
import java.net.URL;
import java.util.Hashtable;

public class MultiFilter {
    public static void main(String[] astrArgs) {
        // Create main window for the application.
        MultiFilterFrame f = new MultiFilterFrame(
                "MultiFilter Image Filter Lab" );
        f.resize(300, 200);
        f.show();

        // Single program parameter may be a file name or
        // a URL to an image file. Load the image. If no
        // parameter given, the loader thread will do
```

```
        // nothing.
        ImageLoaderThread t1 = new
         ImageLoaderThread( astrArgs[0], f );
        if( null != t1 ) t1.start();

        return;
    }

}
```

**3.** The application's main window is a MultiFilterFrame object. Its job is to display a filtered image (if one has been loaded), and to react to user menu selections. Here we declare the MultiFilterFrame class, its internal member variables, and its constructor:

```
class MultiFilterFrame extends Frame {
    Image _imgBase = null;
    Image _imgWorkspace = null;
    WorkspaceCanvas _canvas = new WorkspaceCanvas();
    Label _labelFeedback = new Label( "Initializing…" );
    boolean _fActiveSelection;
    ConfiguredFilterFactory _factoryActive = null;

    // Two hashtables to lookup menu string->factory
    // and factory->last configuration.
    Hashtable _hashMonikerToFactory = new Hashtable();
    Hashtable _hashFactoryToConfig = new Hashtable();

    public MultiFilterFrame(String strTitle) {
        super(strTitle);

        // Create the menu shell. That is, the whole menu should
        // look like this:
        // Image
        // + Open…
        // + <separator>
        // + Refresh
        // + <separator>
        // + Exit
        // Filter
        // + Apply to Image
        // + Apply to Region
        // + <separator>
        // + Null Filter
        // + Contrast Filter
        // + Invert Filter
        Menu menuImage = new Menu( "Image" );
            menuImage.add( "Open…" );
            menuImage.addSeparator();
            menuImage.add( "Refresh" );
            menuImage.addSeparator();
            menuImage.add( "Exit" );

        Menu menuFilter = new Menu( "Filter" );
            menuFilter.add( "Apply to Image" );
            menuFilter.add( "Apply to Region" );
            menuFilter.addSeparator();
            menuFilter.add( "Null Filter" );
```

```
            menuFilter.add( "Contrast Filter" );
            menuFilter.add( "Invert Filter" );

        MenuBar mb = new MenuBar();
        mb.add(menuImage);
        mb.add(menuFilter);

        setMenuBar(mb);

        updateMenu();

        // Add the entries for the three filter types to the
        // lookup hashes.
        _hashMonikerToFactory.put( "Null Filter",
                new ConfiguredFilterFactory() );
        _hashMonikerToFactory.put( "Contrast Filter",
                new ContrastFilterFactory() );
        _hashMonikerToFactory.put( "Invert Filter",
                new InvertFilterFactory() );

        // Add the feedback label and display canvas as
components
        // of this Frame.
        setLayout(new BorderLayout());
        add("South", _labelFeedback);
        add("Center", _canvas);
    }
}
```

**4.** The ImageLoaderThread class is responsible for loading images from the local file system or from the Internet. The Thread keeps a reference to the MultiFilter application's main window and posts custom Events to that window to indicate the progress of image loading. Here is the code for the ImageLoaderThread class:

```
public class ImageLoaderThread extends Thread {
    URL _urlImage;
    String _strLoc;
    Component _compDeliver;
    Image _img;

    // Custom Event IDs:
    public static final int LOCATION_FORMAT_ERROR = -1;
    public static final int IMAGE_LOADING_ERROR = -2;
    public static final int IMAGE_COMPLETED = -3;
    public static final int IMAGE_LOADING = -4;

    // strLoc: a filename or a URL.
    // comp: Component to notify about the image loading
    // process using custom Events.
    public ImageLoaderThread( String strLoc,
            Component comp ) {
        _strLoc = strLoc;
        _compDeliver = comp;
    }

    public void run() {
        // Tell component that we are starting to load
        // the image
```

```
        _compDeliver.deliverEvent(new Event(_compDeliver,
                IMAGE_LOADING, _strLoc));

        // Attempt to create a URL from the _strLoc.
        // If this fails, assume the string is a local
        // file name. If loading from that file fails,
        // notify the target Component and quit.
        try {
            _urlImage = new URL( _strLoc );
            _img = Toolkit.getDefaultToolkit().
                    getImage(_urlImage);
        } catch (Exception e) {
            _img = Toolkit.getDefaultToolkit().
                    getImage( _strLoc );
        }

        if( null == _img ) {
            _compDeliver.deliverEvent(new Event(
                    _compDeliver,
                    LOCATION_FORMAT_ERROR, _strLoc));
            return;
        }

        // Use a MediaTracker to wait for the Image to be
        // fully loaded.
        MediaTracker mt = new MediaTracker( _compDeliver );
        mt.addImage( _img, 0 );
        try {
            mt.waitForID( 0 );
        } catch(Exception e) {
            _compDeliver.deliverEvent(new Event(
                    _compDeliver,
                    IMAGE_LOADING_ERROR, _strLoc));
            return;
        }

        // Deliver completed image to the component.
        _compDeliver.deliverEvent(new Event(
                _img, IMAGE_COMPLETED, _strLoc));

        return;
    }
}
```

**5.** The application's main window will receive the custom Event notifications from ImageLoaderThreads. In its handleEvent implementation, the MultiFilterFrame must process these messages, as well as any WINDOW_DESTROY messages. Here is the code for the Event handling (i.e., MultiFilterFrame.handleEvent):

```
public boolean handleEvent(Event evt) {
    switch( evt.id ) {
        case ImageLoaderThread.LOCATION_FORMAT_ERROR:
        case ImageLoaderThread.IMAGE_LOADING_ERROR:
            _labelFeedback.setText( "Error loading: " +
                    (String)evt.arg );
            return true;
```

```
case ImageLoaderThread.IMAGE_COMPLETED:
    _labelFeedback.setText( "Image loaded: " +
            (String)evt.arg );
    _imgBase = (Image)evt.target;

    // Create the Workspace, displayed by
    // canvas. Copy base image to it.
    _imgWorkspace = createImage(
            _imgBase.getWidth(this),
            _imgBase.getHeight(this));
    Graphics g = _imgWorkspace.getGraphics();
    g.drawImage(_imgBase, 0, 0, this);
    g.dispose();

    // Set the canvas' image to display.
    _canvas.clearSelRect();
    _canvas.clearImage();
    _canvas.setImage(_imgWorkspace);

    // Resize frame to fit new image size.
    resize(preferredSize());

    // update the menu.
    updateMenu();

    return true;

case ImageLoaderThread.IMAGE_LOADING:
    _labelFeedback.setText( "Loading image: " +
            (String)evt.arg );
    return true;

case Event.WINDOW_DESTROY:
    if( evt.target instanceof Window ) {
        ((Window)evt.target).dispose();
        if( (Window)evt.target == this )
            dispose();
            System.exit(0);
            return true;
    }
    break;

case WorkspaceCanvas.SELECTION_CLEARED:
    _fActiveSelection = false;
    updateMenu();
    return true;

case WorkspaceCanvas.SELECTION_CHANGED:
    _fActiveSelection = true;
    updateMenu();
    return true;

case OpenObjectDialog.OPENOBJECTDIALOG_INPUT:
    ImageLoaderThread t =
        new ImageLoaderThread((String)evt.arg,
            this);
    t.start();
```

```
            return true;

        case ConfiguredFilterFactory.
                    CONFIGURATION_UPDATE:
            System.out.println( "Update to config: " +
                evt.arg );
            _hashFactoryToConfig.put(_factoryActive,
                    evt.arg);
            return true;

        default:
            return super.handleEvent(evt);
     }

     return super.handleEvent(evt);
}
```

**6.** The MultiFilterFrame's action method is where menu click Events are handled. Here is the code:

```
// action() is where action events are funneled.
public boolean action(Event evt, Object target) {
    // Non-menu item actions are ignored by
    // this program.
    if( !( evt.target instanceof MenuItem ) )
        return false;

    // "Open…" means we should open a new image.
    // Notification of user's input is passed through
    // deliverEvent().
    if( "Open…" == (String)evt.arg ) {
       OpenObjectDialog dlg =
               new OpenObjectDialog(this, "Image");
       dlg.show();
       return true;
    }

    // Handle "Exit" menu action by posting a
    // WINDOW_DESTROY to this Frame.
    if( "Exit" == (String)evt.arg ) {
        deliverEvent(new Event(this,
                Event.WINDOW_DESTROY, null));
        return true;
    }

    // "Refresh" simply redraws the base image to the
    // Workspace, getting rid of all previous
    // filterings to the image or any region of the
    // image. Also gets rid of any selection rectangle.
    if( "Refresh" == (String)evt.arg ) {
        _imgWorkspace = createImage(
                _imgBase.getWidth(this),
                _imgBase.getHeight(this));
        Graphics g = _imgWorkspace.getGraphics();
        g.drawImage(_imgBase, 0, 0, this);

        // Set the canvas' image to display.
        _canvas.clearSelRect();
```

```
    _canvas.clearImage();
    _canvas.setImage(_imgWorkspace);

    // Resize frame to fit new image size.
    resize(preferredSize());

    // update the menu.
    updateMenu();

    return true;
}

// "Apply To Image" causes the entire image to be
// sent through the current active filter.
if( "Apply to Image" == (String)evt.arg ) {
    Object objConfig =
        _hashFactoryToConfig.get(_factoryActive);
    Image imgNew =
        createImage(new FilteredImageSource(
         _imgWorkspace.getSource(),
         _factoryActive.createFilter(objConfig)));

    // draw filtered version of selection to
    // workspace image.
    Graphics g = _imgWorkspace.getGraphics();
    g.drawImage(imgNew, 0, 0, this);
    g.dispose();

    // force canvas to display updated image.
    _canvas.repaint();
    return true;
}

// "Apply to Region" causes a filtered version
// of the selection rectangle to be created and drawn
// over the selection rectangle on the display image.
if( "Apply to Region" == (String)evt.arg ) {
    System.out.println( "Applying to Region…" );

    Object objConfig =
        _hashFactoryToConfig.get(_factoryActive);
    Rectangle r = _canvas.getSelRect();

    // create filtered version of the
    // selection rectangle.
    Image imgRegion = createImage(
        new FilteredImageSource(
            _imgWorkspace.getSource(),
            new CropImageFilter(r.x, r.y, r.width,
                        r.height)));
    Image imgFiltered = createImage(
        new FilteredImageSource(
            imgRegion.getSource(),
            _factoryActive.createFilter(objConfig)));

    // draw filtered version of selection to
    // workspace image.
```

```
        Graphics g = _imgWorkspace.getGraphics();
        g.drawImage(imgFiltered, r.x, r.y, this);
        g.dispose();

        // force canvas to display updated image.
        _canvas.repaint();

        return true;
    }

    // Attempt to look up a filter factory in the hashes
    // matching this menu item, and create a
    // configuration for that factory.
    ConfiguredFilterFactory factory =
        (ConfiguredFilterFactory)
            _hashMonikerToFactory.get((String)evt.arg);
    if( null != factory ) {
        Object objOldConfig =
                _hashFactoryToConfig.get( factory );
        Object objNewConfig =
            factory.createConfiguration( this,
                    objOldConfig );
        _hashFactoryToConfig.put( factory,
                    objNewConfig );
        _factoryActive = factory;

        // update the menu
        updateMenu();

        return true;
    }

    // Unrecognized menu item, just let it go.
    return false;
}
```

**7.** The main frame must insure that only appropriate menu items are enabled. For example, the "Applet to Image" menu item would be inappropriate if no images were loaded into the application. Whenever a user action is detected which may affect the state of one or more menu items, MultiFilterFrame.updateMenu is called to update the state of the various menu items. Here is the code for that method:

```
// updateMenu() is called whenever the program state
// has changed in such a way that any of the menu
// items may become enabled/disabled, etc.
private synchronized void updateMenu() {
    // get all the menu items into local references.
    MenuBar mb = getMenuBar();
        Menu menuImage = mb.getMenu(0);
            MenuItem miOpen = menuImage.getItem(0);
            MenuItem miRefresh = menuImage.getItem(2);
            MenuItem miExit = menuImage.getItem(4);
        Menu menuFilter = mb.getMenu(1);
            MenuItem miApplyToImage =
                    menuFilter.getItem(0);
            MenuItem miApplyToRegion =
```

```
                menuFilter.getItem(1);
            MenuItem miNullFilter =
                menuFilter.getItem(3);
            MenuItem miContrastFilter =
                menuFilter.getItem(4);
            MenuItem miRotateFilter =
                menuFilter.getItem(5);

    // Image/Open is always available.

    // Image/Refresh is only available if the base
    // image is not null.
    miRefresh.enable( null != _imgBase );

    // Image/Exit is always available.

    // Filter/<filter> is not available if no image is
    // loaded.
    miNullFilter.enable( null != _imgBase );
    miContrastFilter.enable( null != _imgBase );
    miRotateFilter.enable( null != _imgBase );

    // Filter/Apply to Image is not available if no
    // image is loaded, or if no filter is active.
    miApplyToImage.enable( (null != _imgBase) &&
            (null != _factoryActive) );

    // Filter/Apply To Region is only available if
    // image is loaded, and a region is selected, and
    // a filter is active.
    miApplyToRegion.enable( (null != _imgBase) &&
            _fActiveSelection &&
            (null != _factoryActive) );
}
```

**8.** The Workspace image, which is the filtered copy of the loaded image, is displayed by a WorkspaceCanvas class object. This Canvas is responsible for displaying the image, and for tracking user mouse actions so that the user can select a rectangle of the image for filtering. Here is the declaration of the WorkspaceCanvas class, its member variables, and its constructor:

```
class WorkspaceCanvas extends Canvas {
    Image _imgDisplay = null;
    Rectangle _rectSelection = null;
    boolean _fTrackingSelection = false;
    Point _ptSelectionOrigin = null;

    public static final int SELECTION_CLEARED = -10;
    public static final int SELECTION_CHANGED = -11;

    public WorkspaceCanvas() {}
}
```

**9.** The WorkspaceCanvas class implements methods to get, set, and delete the Image object it is supposed to display:

```
private synchronized Image getDisplayImage() {
    return _imgDisplay;
}
```

```
public synchronized void clearImage() {
    _imgDisplay = null;
    return;
}

public synchronized void setImage(Image img) {
    if( img == _imgDisplay )
        System.out.println( "Setting to same image." );
    else
        System.out.println( "Setting to new image." );

    _imgDisplay = img;
    resize( img.getWidth(this), img.getHeight(this) );
    repaint();
    return;
}
```

**10.** The WorkspaceCanvas' update and paint methods are implemented to avoid flicker and to draw the image to display:

```
// Avoid flicker in paint() by overriding update
// not to erase background.
public void update(Graphics g) {
    paint(g);
}

// Paint draws the display image and the active
// selection, if there is any.
public synchronized void paint(Graphics g) {
    Image img;

    if( null == (img = getDisplayImage()) )
        return;

    // First, draw the display image.
    g.setPaintMode();
    g.drawImage(img, 0, 0, this);

    // Now, draw selection rectangle, if there is one.
    drawSelRect(g);
    return;
}
```

**11.** The WorkspaceCanvas uses XOR mode to draw the selection rectangle. This ensures that no matter what color the display image is, the boundaries of the selection rectangle are always visible to the user. Here's the code for the drawSelRect method:

```
// Draw the selection rectangle. Use XOR mode so the selection
// rectangle is an XOR drawing.
private void drawSelRect(Graphics g) {
    Rectangle rectSel = getSelRect();
    if( null != rectSel ) {
        g.setColor( Color.white );
        g.setXORMode( Color.black );
        g.drawRect(rectSel.x, rectSel.y, rectSel.width,
             rectSel.height);
    }
```

```
    return;
}
```

**12.** The user defines the selection rectangle using mouse clicks and drags. The WorkspaceCanvas keeps track of these mouse Events through mouseDown and mouseDrag. Here's the code for those methods:

```
// mouseDown() and mouseDrag() notification methods
// are used to track the selection region.
public synchronized boolean mouseDown(Event evt,
        int x, int y) {
    _ptSelectionOrigin = new Point(x, y);
    clearSelRect();
    repaint();
    return true;
}

public synchronized boolean mouseDrag(Event evt,
        int x, int y) {
    if( (null == _ptSelectionOrigin) ||
            (null == _imgDisplay) )
        return false;

    Dimension d = new Dimension(
            _imgDisplay.getWidth(this),
            _imgDisplay.getHeight(this));
    if( ((x<=_ptSelectionOrigin.x) ||
            (x > d.width)) ||
        ((y<=_ptSelectionOrigin.y) ||
            (y > d.height)) )
        return true;

    setSelRect(new Rectangle(_ptSelectionOrigin.x,
            _ptSelectionOrigin.y, x -
            _ptSelectionOrigin.x,
            y - _ptSelectionOrigin.y ));
    return true;
}
```

**13.** The WorkspaceCanvas also defines helper methods to get, set, and clear the boundaries of the selection rectangle. Here is the code:

```
public synchronized Rectangle getSelRect() {
    return _rectSelection;
}

public synchronized void clearSelRect() {
    _rectSelection = null;
    deliverEvent(new Event(this, SELECTION_CLEARED, null));
    repaint();
    return;
}

public synchronized void setSelRect(Rectangle rectNewSel) {
    _rectSelection = rectNewSel;
    deliverEvent(new Event(this, SELECTION_CHANGED,
       _rectSelection));
    repaint();
    return;
}
```

**14.** The OpenObjectDialog simply gathers a string from the user. This string can be either a filename on the local file system or a URL pointing to a file somewhere on the Internet. An OK and a Cancel Button are also provided. The MultiFilterFrame uses an OpenObjectDialog whenever the user clicks on the "Open…" menu item. Here is the code for that class:

```
class OpenObjectDialog extends Dialog {
    TextField _tfInput;

    public static final int OPENOBJECTDIALOG_INPUT = -20;

    public OpenObjectDialog(Frame frameParent,
            String strType) {
        // Make this a model dialog.
        super(frameParent, "Open " + strType, true);
        String strInstructions =
                "File pathname or URL for " + strType;
        setResizable(false);

        // Create a label with instructions, a text box
        // for user input, an OK and a Cancel button, and
        // a panel to place OK and Cancel buttons on.
        Label l = new Label(strInstructions);
        _tfInput = new TextField();
        Panel p = new Panel();

        p.setLayout(new FlowLayout());
        Button ok = new Button( "OK" );
        Button cancel = new Button( "Cancel" );
        p.add( ok );
        p.add( cancel );

        // set layout for Dialog to be a grid with 1 col,
        // 3 rows.
        setLayout(new GridLayout(3, 1));
        add( l );
        add( _tfInput );
        add( p );

        resize(200, 150);
    }

    // Button events are caught using action().
    // Only the OK button has any actions associated with
    // it, and then only if the input field is not empty.
    // Cancel results in the window being destroyed.
    public boolean action(Event evt, Object what) {
        if( !(evt.target instanceof Button) )
            return false;

        if( "OK" != (String)what ) {
            if( "Cancel" == (String)what ) {
                dispose();
                return true;
            }
            return false;
        }
```

```
        if( (null == _tfInput.getText()) ||
                ("" == _tfInput.getText()) )
            return false;

        String strInput = _tfInput.getText();
        deliverEvent(new Event(this,
                OPENOBJECTDIALOG_INPUT, strInput));
        dispose();
        return true;
    }
}
```

**15.** Create a new file named "ConfiguredFilterFactory.java". This file holds the declaration of the ConfiguredFilterFactory class. The ConfiguredFilterFactory class is a base class. In the MultiFilter application, both the ContrastFilterFactory and the InvertFilterFactory are based on this class. In general, a ConfiguredFilterFactory should be able to create a new ImageFilter that should be configured according to the value of an Object. Here is the declaration:

```
import java.awt.image.ImageFilter;
import java.awt.Frame;

public class ConfiguredFilterFactory {
    public static final int CONFIGURATION_UPDATE = 40;

    public synchronized Object createConfiguration(
            Frame frameParent, Object objLastConfiguration)
    {
        return new Object();
    }

    public synchronized ImageFilter createFilter(
               Object objConfiguration) {
        return new ImageFilter();
    }
}
```

**16.** Create a new file called "ContrastFilterFactory.java". Start off by declaring the ContrastFilterFactory class, and importing the necessary packages into this file:

```
import java.awt.image.ImageFilter;
import java.awt.image.ColorModel;
import java.awt.image.RGBImageFilter;
import java.awt.*;

public class ContrastFilterFactory extends
        ConfiguredFilterFactory {

    public Object createConfiguration(Frame frameParent,
            Object objLastConfiguration) {
        int nContrast = (null==objLastConfiguration) ? -1 :
                ((Integer)objLastConfiguration).intValue();

        ContrastFilterConfigurationDialog dlg = new
                ContrastFilterConfigurationDialog(
                        frameParent, nContrast);
        dlg.show();
```

```
        return new Integer(dlg.getContrast());
    }

    public ImageFilter createFilter(Object objConfig) {
        return new ContrastFilter(
                ((Integer)objConfig).intValue());
    }
}
```

**17.** A ContrastFilter is a type of RGBImageFilter. In its filterRGB implementation, it increases or decreases the contrast of a particular pixel, based on its constructor parameters. Here is the code:

```
class ContrastFilter extends RGBImageFilter {
    int _nContrast;

    public ContrastFilter(int nContrast) {
        canFilterIndexColorModel = true;
        _nContrast = nContrast;
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = (rgb & 0x00ff0000) >> 16;
        int g = (rgb & 0x0000ff00) >> 8;
        int b = rgb & 0x000000ff;

        double dContrastRads = (double)2 * Math.PI *
                ((double)_nContrast/(double)360);
        double dTan = Math.tan(dContrastRads);

        int rNew = Math.min(255, Math.max(0,
                (int)(dTan*(r - 128) + 128)));
        int gNew = Math.min(255, Math.max(0,
                (int)(dTan*(g - 128) + 128)));
        int bNew = Math.min(255, Math.max(0,
                (int)(dTan*(b - 128) + 128)));

        return (rNew << 16) | (gNew << 8) | bNew |
                (rgb & 0xff000000);
    }
}
```

**18.** The ContrastFilterConfigurationDialog is used to make the user select a contrast parameter for a new ContrastImageFilter. It uses a Scrollbar as its principal input device. A ContrastDisplayCanvas is used in the dialog to display the contrast function as indicated by the state of the Scrollbar. Here is the code for those two classes:

```
class ContrastFilterConfigurationDialog extends Dialog {
    int _nContrast = 45; //0-89 value: angle of contrast
                         // function line.
    Scrollbar _scrollContrast = new Scrollbar(
            Scrollbar.HORIZONTAL, 45, 10, 0, 89);
    ContrastFunctionDisplayCanvas _canvas =
            new ContrastFunctionDisplayCanvas(_nContrast);

    public ContrastFilterConfigurationDialog(
            Frame frameParent, int nContrast) {
        super(frameParent, "Contrast Filter Configuration",
```

```
                false);

        if( -1 != nContrast )
            _nContrast = nContrast;

        add( "South", _scrollContrast);
        add( "Center", _canvas );

        resize(200, 200);
    }

    public int getContrast() {
        return _nContrast;
    }

    // Implementation of handleEvent so dialog can handle
    // scrollbar updates.
    public boolean handleEvent(Event evt) {
        switch(evt.id) {
            case Event.SCROLL_LINE_UP:
            case Event.SCROLL_LINE_DOWN:
            case Event.SCROLL_PAGE_UP:
            case Event.SCROLL_PAGE_DOWN:
            case Event.SCROLL_ABSOLUTE:
                _nContrast = _scrollContrast.getValue();
                getParent().deliverEvent(new Event(
                  getParent(),
                  ConfiguredFilterFactory.
                      CONFIGURATION_UPDATE,
                  new Integer(_nContrast)));
                _canvas.setContrast(_nContrast);
                return true;
        }

        // all other events passed on.
        return false;
    }
}

class ContrastFunctionDisplayCanvas extends Canvas {
    int _nContrast;

    public ContrastFunctionDisplayCanvas(int nContrast) {
        setContrast(nContrast);
    }

    public void setContrast(int nContrast) {
        _nContrast = nContrast;
        setBackground( Color.black );
        setForeground( Color.white );
        repaint();
    }

    public void paint(Graphics g) {
        // Draw min/max horiz. lines at 5/6 and 1/6 the way
        // up the component.
        Rectangle r = bounds();
```

```
        int nMax = (5 * r.height) / 6;
        int nMin = r.height / 6;

        g.drawLine(0, nMax, r.width, nMax);
        g.drawLine(0, nMin, r.width, nMin);

        // Draw contrast line. To keep it between min
        // and max lines, we'll constrict the clipping rectangle
        // to between those lines.
        g.clipRect(0, nMax, r.width, nMin);

        double dContrastRads = (double)2 * Math.PI *
                ((double)_nContrast/(double)360);
        double dTan = Math.tan(dContrastRads);

        if( 0 != dTan ) {
            int w = (int)(((double)nMax -
                    (double)nMin)/dTan);
            g.drawLine(r.width/2 - w/2, nMin,
                    r.width/2 + w/2, nMax);
        }
        else
            g.drawLine(0, r.height/2, r.width, r.height/2);
    }
}
```

**19.** Create a new file named "InvertFilterFactory.java". Start off by declaring the InvertFilterFactory. This is very similar to the ContrastFilterFactory, except that it creates InvertFilters, and gets its configuration from InvertFilterConfigurationDialogs. Here is the code for the InvertFilterFactory:

```
import java.awt.image.ImageFilter;
import java.awt.image.ColorModel;
import java.awt.*;

public class InvertFilterFactory extends
        ConfiguredFilterFactory {

    public Object createConfiguration(Frame frameParent,
            Object objLastConfiguration) {
        int flags = (null==objLastConfiguration) ? -1 :
                ((Integer)objLastConfiguration).intValue();

        InvertFilterConfigurationDialog dlg = new
            InvertFilterConfigurationDialog(frameParent,
                    flags);
        dlg.show();

        return new Integer(dlg.getFlags());
    }

    public ImageFilter createFilter(Object objConfig) {
        System.out.println( "Creating filter with: " +
                objConfig );
        return new InvertFilter(
                ((Integer)objConfig).intValue());
    }
}
```

**20.** The InvertFilter can be configured to invert an input image along either the X or Y axis, or both. Its setPixels implementation works by swapping individual pixel values to invert the image. Here is the code for that class:

```
class InvertFilter extends ImageFilter {
    Dimension _dim = null;
    int _nFlags = 0;

    public static final int HORIZONTAL = 1;
    public static final int VERTICAL = 2;

    public InvertFilter(int flags) {
        _nFlags = flags;
    }

    public void setDimensions(int width, int height) {
        _dim = new Dimension(width, height);
        consumer.setDimensions(width, height);
        return;
    }

    public void setPixels(int x, int y, int w, int h,
            ColorModel cm, byte[] pixels, int off,
            int scansize) {
        // 1. transform rectangle to destination rectangle
        int xNew = (0!=(_nFlags & HORIZONTAL)) ?
                _dim.width-x-w : x;
        int yNew = (0!=(_nFlags & VERTICAL)) ?
                _dim.height-y-h : y;

        // 2. invert pixels in the pixels[] array. This
        // involves transforming the pixels' array about
        // the axis of inversion. Done in two steps:
        // HORIZONTAL, then VERTICAL.
        if( 0 != (_nFlags & HORIZONTAL) ) {
            for( int j=0 ; j<h ; j++ ) {
                for( int i=0 ; i<w/2 ; i++ ) {
                  byte bTemp = pixels[(j*scansize)+i+off];
                  pixels[(j*scansize)+i+off] =
                      pixels[(j*scansize)+(w+x-i-1)+off];
                  pixels[(j*scansize)+(w+x-i-1)+off]=bTemp;
                }
            }
        }

        if( 0 != (_nFlags & VERTICAL) ) {
            for( int i=0 ; i<w ; i++ ) {
                for( int j=0 ; j<h/2 ; j++ ) {
                  byte bTemp=pixels[(j*scansize)+i+off];
                  pixels[(j*scansize)+i+off] =
                      pixels[((h+y-j-1)*scansize)+i+off];
                  pixels[((h+y-j-1)*scansize)+i+off]=bTemp;
                }
            }
        }

        // send the transformed pixels on to the consumer.
```

```
        consumer.setPixels(xNew, yNew, w, h, cm, pixels,
                off, scansize);
    }

    public void setPixels(int x, int y, int w, int h,
            ColorModel cm, int[] pixels, int off,
            int scansize) {
    }
}
```

**21.** The InvertFilterConfigurationDialog is a simple dialog with two checkboxes, one each of horizontal and vertical invert. Here is the code:

```
class InvertFilterConfigurationDialog extends Dialog {
    int _nFlags = InvertFilter.VERTICAL |
            InvertFilter.HORIZONTAL;
    Checkbox _cbHoriz = new Checkbox("Invert horizontal");
    Checkbox _cbVert = new Checkbox("Invert vertical");
    // Layout of this dialog is one row of controls:
    //   + invert horizontal
    //   + invert vertical
    public InvertFilterConfigurationDialog(
            Frame frameParent, int flags) {
        super(frameParent, "Invert Filter Configuration",
            false);
        if( -1 != flags )
            _nFlags = flags;

        _cbHoriz.setState(0 !=
                (_nFlags & InvertFilter.HORIZONTAL));
        _cbVert.setState(0 !=
                (_nFlags & InvertFilter.VERTICAL));

        setLayout(new FlowLayout());
        add( _cbHoriz );
        add( _cbVert );

        resize(250, 75);
    }

    public int getFlags() {
        return _nFlags;
    }

    // Capture the checkbox events, and change the
    // configuration as the user wishes. Report all
    // configuration changes to parent window as
    // ConfiguredFilterFactory.CONFIGURATION_UPDATE Events.
    public boolean action(Event evt, Object what) {
        if( !(evt.target instanceof Checkbox) )
            return false;

        if( (Checkbox)evt.target == _cbHoriz ) {
            if( _cbHoriz.getState() )
                _nFlags |= InvertFilter.HORIZONTAL;
            else
                _nFlags &= ~InvertFilter.HORIZONTAL;
```

```
                getParent().deliverEvent(new Event(getParent(),
                    ConfiguredFilterFactory.
                        CONFIGURATION_UPDATE,
                    new Integer(_nFlags)));
                return true;
            }

            if( (Checkbox)evt.target == _cbVert ) {
                if( _cbVert.getState() )
                    _nFlags |= InvertFilter.VERTICAL;
                else
                    _nFlags &= ~InvertFilter.VERTICAL;

                getParent().deliverEvent(new Event(getParent(),
                    ConfiguredFilterFactory.
                        CONFIGURATION_UPDATE,
                    new Integer(_nFlags)));
                return true;
            }

            // all other actions passed along.
            return false;
        }
    }
```

**22.** Compile the project. In the MultiFilter directory, execute this command to compile the MultiFilter project using the JDK's javac compiler:

```
> javac MultiFilter.java
```

Correct any errors in typing, which will be reported as errors on the command line.

**23.** Run the MultiFilter application. Provide an image's full pathname on your local system, or an image URL as the first and only argument to this program. Use this command line:

```
> java MultiFilter <your-image-file-or-URL>
```

## How It Works

A lot of MultiFilter's code implements the user-interface of the application. You are encouraged to study the MultiFilter application's user-interface model and techniques, but this discussion will be restricted to the AWT Image Processing functionalities demonstrated by MultiFilter and how you can extend MultiFilter by adding your own ImageFilter objects its interface.

### *Loading and Storing the Base Image*

At the heart of MultiFilter are two Image objects: *MultiFilterFrame._imgBase* is a pristine copy of an Image as it was loaded from a local file system file, or from a network URL. MultiFilterFrame.*_imgWorkspace* is a modified version of the base image.

Whenever a filter is applied to either the entire displayed image, or to a region of it, it is *_imgWorkspace* which is modified.

The base image is loaded and prepared for use by the ImageLoaderThread object. The ImageLoaderThread is designed to load Image objects from a local file system file, or from a network URL, on behalf of a Component. All downloading and preparing of the Image are done by a background Thread. When the Image has been fully downloaded, the ImageLoaderThread delivers the Image, as the *arg* member of a custom Event, to the Component. The Event ID is ImageLoaderThread.IMAGE_COMPLETE.

### *ConfiguredFilterFactory*

To make MultiFilter as extensible as possible, the ConfiguredFilterFactory was developed to make creating and configuring generic ImageFilter objects a simple operation for the MultiFilter application. The jobs of a ConfiguredFilterFactory are

- To perform user-interaction in a modeless dialog, allowing the user to dynamically modify the configuration of an ImageFilter. createConfiguration is overridden by classes derived from ConfiguredFilterFactory to create such a modeless dialog. Enough information to describe a filter configuration must be able to be stored in a single Object instance, though since all objects in Java ultimately derive from Object, this is not a constraint on the amount of information you can cull from the user to configure a filter.
- To create configured ImageFilter objects, the configuration reflecting the user-input to the modeless dialog is passed to createFilter. This method should return an ImageFilter object initialized using the configuration information the user input into the modeless dialog.

The default implementation of ConfiguredFilterFactory creates objects of class ImageFilter. These objects are null-filters. That is, when applied to an Image, the filtered output looks exactly the same as the unfiltered input. Furthermore, no user-interaction is necessary for such an object, since the ImageFilter class takes no parameters. Thus the default implementations of createConfiguration and createFilter are pretty simple.

### *The ContrastFilter*

The ContrastFilter is an RGBImageFilter. The filter is created with a “contrast” parameter, which defines how much contrast the output Image should have in it. This contrast parameter is interpreted as a degree which defines the slope of the linear relationship between input and output RGB color. If the angle is low, then the slope is near-horizontal, and the output RGB color components are right around 128. If, on the other hand, the angle is high then the slope of the linear relationship is high, and slight differences in the brightness between pixels is magnified in the output image. Figure 8-12 shows the ContrastFilterConfigurationDialog, which is the modeless dialog the user uses to modify the contrast settings.


**Figure 8-12** The ContrastFilterConfigurationDialog

### *The InvertFilter*

The InvertFilter is an ImageFilter-derived class. Because the filtered output of each pixel of this filter is dependent on other pixels in the image, the InvertFilter could not be derived from RGBImageFilter.

The internal state of an InvertFilter object is stored as two flags: HORIZONTAL and VERTICAL, which are both set when the filter is created by parameters to the InvertFilter constructor. The InvertFilter.setPixels() method flips each individual rectangle of the input image in the filtered version of the image. Two separate flipping operations may be necessary: HORIZONTAL flipping, and VERTICAL flipping, according to the internal state of the InvertFilter.

"Flipping" of a rectangle involves tranforming the coordinates of the rectangle to the output Image. This transformation ensures the width and height of the rectangle are preserved in the output image, but the origin of the rectangle in the Image is moved. The individual pixel values of the rectangle are flipped symmetrically about either the X or Y axis, according to the internal state of the InvertFilter object. Figure 8-13 illustrates how a rectangle of pixel data is transformed for the filtered output.

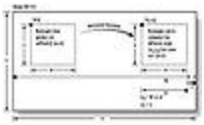

**Figure 8-13** InvertFilter translates rectangles so the rectangle covers a symmetrical area with respect to the horizontal or vertical axis

Once the pixel data has been flipped in the *pixels[]* array, and the origin of the rectangle has been transformed for the output Image, the pixel data is passed through to the filter's consumer.

### *Applying Filters to the Workspace Image*

A FilteredImageSource object is used to apply the currently selected filter to the whole Workspace image, or to a selected region of it. This is handled in the MultiFilterFrame.action() method, where the MenuItem selection action is detected by the MultiFilterFrame.

In response to the "Apply to Image" menu item being selected, MultiFilter creates a new Image object which is the output of applying the currently selected filter to the entire

Workspace image. This new Image object then paints over the contents of the Workspace image, thus essentially replacing the Workspace image with itself.

The WorkspaceCanvas displays the current contents of the Workspace image, so that as soon as the filter is applied and the Workspace is filled with a copy of the filtered image, the Workspace is displayed on the screen.

In response to the "Apply to Region" menu item being selected, MultiFilter first creates a copy of just the selected region using a CropImageFilter and a FilteredImageSource. A new image is created which is the result of sending the cropped image selection through the currently selected filter. This filtered version of the selected region is then painted onto the Workspace image at precisely the same location as the selected region. When the Workspace is redisplayed, the filtered version of the selected region replaces the original pixel data there.

## Adding Your Own ImageFilters to MultiFilter

MultiFilter was designed to make adding new filters as easy as possible. Follow these steps to add a new filter to MultiFilter:

**1.** Create your ImageFilter.
**2.** Create a modeless Dialog so that the user of MultiFilter can configure your ImageFilter. For example, if your filter requires, say, a "zoom factor" in its constructor, then create a Dialog that allows the user to pick a "zoom factor". This Dialog object should deliver a ConfiguredFilterFactory.CONFIGURATION_UPDATE Event to its parent (result of getParent()) frame whenever the user modifies the values in the Dialog. See ContrastFilterConfigurationDialog in the MultiFilter application (Step 18) for an example.
**3.** Create a ConfiguredFilterFactory for your ImageFilter class. The overridden version of createConfiguration should create and show a new instance of the Dialog you defined in Step 2. The overridden version of createFilter should create and return a new instance of the ImageFilter you defined in Step 1, configured using the configuration Object passed as a parameter. See ContrastFilterFactory of the MultiFilter application (Step 16) for an example.
**4.** Add code to MultiFilterFrame constructor to add a menu item for your new filter. In the same constructor method, you should also add an entry to Hashtable *_hashMonikerToFactory*. For example, if I created a filter class called MakeImagePretty and corresponding factory and configuration objects called MakeImagePrettyFactory and MakeImagePrettyConfigurationDialog, then I would change MultiFilterFrame's constructor like so:

```
public MultiFilterFrame(String strTitle) {
        super(strTitle);

        // Create the menu shell. That is, the whole menu should
```

```
        // look like this:
        // Image
        // + Open…
        // + <separator>
        // + Refresh
        // + <separator>
        // + Exit
        // Filter
        // + Apply to Image
        // + Apply to Region
        // + <separator>
        // + Null Filter
        // + Contrast Filter
        // + Invert Filter
        // + Make Pretty Filter
        Menu menuImage = new Menu( "Image" );
            menuImage.add( "Open…" );
            menuImage.addSeparator();
            menuImage.add( "Refresh" );
            menuImage.addSeparator();
            menuImage.add( "Exit" );

        Menu menuFilter = new Menu( "Filter" );
            menuFilter.add( "Apply to Image" );
            menuFilter.add( "Apply to Region" );
            menuFilter.addSeparator();
            menuFilter.add( "Null Filter" );
            menuFilter.add( "Contrast Filter" );
            menuFilter.add( "Invert Filter" );
            menuFilter.add( "Make Pretty Filter" );

        MenuBar mb = new MenuBar();
        mb.add(menuImage);
        mb.add(menuFilter);

        setMenuBar(mb);

        updateMenu();

        // Add the entries for the three filter types to the
        // lookup hashes.
        _hashMonikerToFactory.put( "Null Filter",
                new ConfiguredFilterFactory() );
        _hashMonikerToFactory.put( "Contrast Filter",
                new ContrastFilterFactory() );
        _hashMonikerToFactory.put( "Invert Filter",
                new InvertFilterFactory() );
        _hashMonikerToFactory.put( "Invert Filter",
                new MakeImagePrettyFactory() );

        // Add the feedback label and display canvas as
components
        // of this Frame.
        setLayout(new BorderLayout());
        add("South", _labelFeedback);
        add("Center", _canvas);
    }
```

```
}
```

That's it! Recompile MultiFilter and try out your filter on a test image.

# Chapter 9
# AWT Peer Interfaces

This chapter introduces the concepts behind the peer interfaces of the Java Abstract Windowing Toolkit (AWT). As a Java application developer you will never need to use any of the peer classes. The interfaces described in this chapter will be of interest mainly to programmers who are porting the Java Development Kit (JDK) to a new operating system. This introduction gives you an insight into the object-oriented design of the peer interfaces and describes how they help reduce the effort needed to port the Java AWT to another operating system.

The individual characteristics of the graphics toolkits supported on different operating systems make the task of implementing a portable windowing toolkit difficult. Separating the user's perspective on the windowing abstraction from the operating system's perspective poses a problem that involves a lot of programming effort when moving the toolkit from one platform to another. If the actual implementation of the toolkit is decoupled from the user's perspective, then porting the toolkit as a whole is a lot simpler, as only the implementation module has to be changed to support the new operating system platform. The peer interfaces of the Java AWT do exactly that. They decouple the abstractions of the AWT classes (such as Button, List, Label, and so on) from their implementation on a specific windowing platform (such as the Windows95 operating system or the Motif windowing toolkit available on Unix systems).

The appearance of a graphical user interface component differs from toolkit to toolkit. The common problem faced with developing applications using the windowing toolkits available today is that the user has to relearn the look and feel of the graphical components when the application is run on a different platform. The Java design team designed the AWT in such a way that the graphical components would preserve the look-and-feel of the underlying operating system's native windowing toolkit. This is one reason why Java has been accepted as a language for cross-platform application development.

Figure 9-1 shows two views of a Java application that includes a label, a button, and a checkbox. The window on the left shows the application running on the Windows95 operating system and the one on the right shows the same application running on the Solaris operating system (using the Motif windowing toolkit).

**Figure 9-1** The same AWT application running on Windows95 and on Solaris (Motif windowing toolkit)

The GUI-specific code is encapsulated in the classes that implement the peer interface. These peer classes serve as the interface between the AWT classes such as Component, Button, Label, Container, and so on, and the underlying operating system's graphics primitives. The AWT classes do not communicate directly with the native windowing toolkit. They create a peer object that serves as a channel of communication between the AWT class and the native windowing toolkit. Each instance of a class (such as Button) is associated with an instance of the corresponding peer class (such as ButtonPeer). In order to port the Java AWT to a windowing toolkit, only the peer interfaces have to be implemented. The higher level of abstraction such as the Component, Button, or List class need not change at all. Figure 9-2 shows where the peer interfaces fit into the overall Java AWT model.

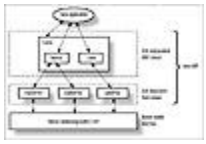

**Figure 9-2** The role of peers in the AWT system model

Each of the peer interfaces described in this chapter has to be implemented to create a GUI-dependent class that performs the task required of the peer. The code for these methods is written in the C language, using the graphics primitives of the native windowing toolkit. Chapter 3 on the Toolkit class and Appendix C on writing native code for Java, provide more information regarding the specifics of implementing native code for a target windowing system.

Due to the licensing issues involved with modifying the source code of the Java Development Kit, this chapter does not contain a project implementation and there are no example entries for individual interfaces or methods. Here is a checklist of the things you have to do in order to extend the peer interfaces described in this chapter to support a target native windowing toolkit.

1. Contact Sun Microsystems with regard to licensing the source code of the Java Development Kit.
2. Identify the graphics primitives used in the target operating system.
3. Implement each of the peer interfaces described in this chapter, using the native methods of the target windowing system. The implementations for the windowing toolkits already supported (Motif, WindowsNT, and Windows95) will help you understand exactly what needs to be done.
4. Extend the java.awt.Toolkit class to create a GUI-dependent class for the target windowing system. This class will include methods to create each of the peer classes implemented in Step 3.
5. Compile and install the classes in the appropriate directories.

The Java language is growing in popularity as the language of choice for cross-platform application development. When porting the Java Development Kit to a new operating system or to a new windowing system, the peer interfaces have to be ported to conform to the application programming interface of the target system. The AWT Peer interfaces clearly separate the platform dependent (physical) abstractions of the windowing toolkit from the user-interface (logical ) abstractions, thus making the job easier for developers who are porting Java.

## AWT Peer Interface Summaries

Table 9-1 summarizes the interfaces described in this chapter. The rest of the chapter presents the detailed descriptions of the methods defined in each of the interfaces.

**Table 9-1** AWT Peer interface descriptions

| Interface | Description |
|---|---|
| ComponentPeer | This interface defines the API between the Component class and the underlying operating system GUI primitives used for querying and modifying the properties associated with an AWT component. |
| ButtonPeer | This interface defines the API between the Button class and the underlying operating system GUI primitives used with button objects |
| CanvasPeer | This interface defines the API between the Canvas class and the underlying operating system GUI primitives used with general-purpose canvas objects. |
| CheckboxPeer | This interface defines the API between the Checkbox class and the underlying operating system GUI primitives used with checkbox objects. |
| ChoicePeer | This interface defines the API between the Choice class and the underlying operating system GUI primitives used for creating and interacting with pop-up menu objects. |
| LabelPeer | This interface defines the API between the Label class and the underlying operating system GUI primitives used for creating and interacting with simple label objects that display a noneditable text string. |
| ListPeer | This interface defines the API between the List class and the underlying operating system GUI primitives used for creating and interacting with graphical objects that contain a scrolling list of text strings. |

| | |
|---|---|
| ScrollbarPeer | This interface defines the API between the Scrollbar class and the underlying operating system GUI primitives used for creating and interacting with scrollbar objects. |
| ContainerPeer | This interface defines the API between the Container class and the underlying operating system GUI primitives used for creating and interacting with graphical objects that can contain other graphical objects. |
| PanelPeer | This interface defines the API between the Panel class and the underlying operating system GUI primitives used for creating and interacting with panel objects. |
| WindowPeer | This interface defines the API between the Window class and the underlying operating system GUI primitives used for creating and interacting with top-level windows that have neither a title bar nor a border. |
| DialogPeer | This interface defines the API between the Dialog class and the underlying operating system GUI primitives used for creating and interacting with dialog box objects. |
| FileDialogPeer | This interface defines the API between the FileDialog class and the underlying operating system GUI primitives used for creating and interacting with dialog box objects that display a list of files and allow the user to select from the list. |
| FramePeer | This interface defines the API between the Frame class and the underlying operating system GUI primitives used for creating and interacting with top-level frame windows that have a title bar and borders and can optionally have a menu bar. |
| MenuComponentPeer | This interface defines the API between the MenuComponent class and the underlying operating system GUI primitives used for creating and interacting with menu objects. |
| MenuBarPeer | This interface defines the API between the MenuBar class and the underlying operating system GUI primitives used for creating and interacting with menu bar objects. |
| MenuItemPeer | This interface defines the API between the MenuItem class and the underlying operating system GUI primitives used for creating and interacting with menu item objects. |
| CheckboxMenuItemPeer | This interface defines the API between the CheckboxMenuItem class and the underlying GUI primitives used for creating and interacting with checkboxes that can be used as menu item objects. |
| MenuPeer | This interface defines the API between the Menu class and the underlying GUI primitives that can be components of a menu bar. |
| TextComponentPeer | This interface defines the API between the TextComponent |

| | |
|---|---|
| | class and the underlying operating system GUI primitives used for creating graphical objects in which text can be edited. |
| TextAreaPeer | This interface defines the API between the TextArea class and the underlying operating system GUI primitives used for displaying and editing multiple lines of text. |
| TextFieldPeer | This interface defines the API between the TextField class and the underlying operating system GUI primitives used for creating graphical objects that allow editing of a single line of text. |

## ComponentPeer

**Purpose**

This interface defines the API between the Component class and the underlying operating system GUI primitives used for querying and modifying the properties associated with an AWT component.

**Syntax**

public interface ComponentPeer extends Object

**Description**

This interface defines the API between the Component class and the underlying operating system GUI primitives used for querying and modifying graphical components. Native code that uses the API of the underlying GUI toolkit to perform these tasks is encapsulated in a GUI-dependent class that implements this interface. Figure 9-3 shows the inheritance hierarchy for the ComponentPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.ComponentPeer;*

**Constructors**

None.

**Parameters**

None.

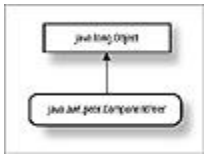

**Figure 9-3** Inheritance hierarchy for the ComponentPeer interface

## checkImage(Image, int, int, ImageObserver)

**InterfaceName**

ComponentPeer

**Purpose**

Returns the status of construction of the specified image object.

**Syntax**

public abstract int checkImage(Image img, int w, int h, ImageObserver o)

**Parameters**

***img***

The Image object whose screen representation is being constructed.

***w***

The width of the image to check the status of.

***h***

The height of the image to check the status of.

***o***

The ImageObserver object that is to be notified as the specified image is being constructed.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to determine the status of construction of the screen representation of the Image object specified as a parameter to this method.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

This method returns an integer value that indicates the status of construction of the Image object as indicated by the specified class that implements the ImageObserver interface.

**See Also**

The checkImage method of the Component class in Chapter 2

## createImage(ImageProducer)

**InterfaceName**

ComponentPeer

**Purpose**

Creates an Image from the producer object which is an instance of a class that implements the ImageProducer interface.

**Syntax**

public abstract Image createImage(ImageProducer producer)

**Parameters**

***producer***

The ImageProducer object from which the image is created.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to create the image from the ImageProducer object.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

The return value of this method is an Image object that represents the image that was produced by the ImageProducer.

**See Also**

The createImage(ImageProducer) method of the Component class in Chapter 2

## createImage(int, int)

**InterfaceName**

ComponentPeer

**Purpose**

Creates an Image of the specified dimensions. This image can be used for updating the screen using the double-buffering technique.

**Syntax**

public abstract Image createImage(int width, int height)

**Parameters**

***width***

The width of the image to create.

***height***

The height of the image to create.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to create the off-screen representation of the Image object with the specified width and height.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

The return value of this method is an Image object that represents the image that can be used for double-buffering.

**See Also**

The createImage(int, int) method of the Component class in Chapter 2

## disable()

**InterfaceName**

ComponentPeer

**Purpose**

Disables the Component (associated with this peer) so that it neither responds to user actions nor generates events.

**Syntax**

public abstract void disable()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to disable the component from responding to events.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The disable method of the Component class in Chapter 2

## dispose()

**InterfaceName**

ComponentPeer

**Purpose**

Frees the resources allocated to this peer object.

**Syntax**

public abstract void dispose()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to free all resources that have been allocated to this peer object. The Component object invokes this method to destroy the ComponentPeer object.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The dispose method of the Component class in Chapter 2

## enable()

**InterfaceName**

ComponentPeer

**Purpose**

Enables the Component object, so that it responds to events.

**Syntax**

public abstract void enable()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment such that the Component object generates events in response to user actions, etc.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The enable method of the Component class in Chapter 2

## getColorModel()

**InterfaceName**

ComponentPeer

**Purpose**

Obtains the RGB values for the ColorModel used to display the Component on the current output device.

**Syntax**

public abstract ColorModel getColorModel()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to determine the Component object's color values in RGB notation.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

This method returns a ColorModel object that describes the RGB color model used to display the component on the output device.

**See Also**

The getColorMethod method of the Component class in Chapter 2; the ColorModel class in Chapter 7

## getFontMetrics(Font)

**InterfaceName**

ComponentPeer

**Purpose**

Obtains information about the properties of the specified font for this Component.

**Syntax**

public abstract FontMetrics getFontMetrics(Font font)

**Parameters**

***font***

The font object whose metrics are to be retrieved.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to determine the various metrics of the font used by the Component object.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

This method returns a FontMetrics object that describes the characteristics of the font used by the component (associated with this peer).

**See Also**

The getFontMetrics method of the Component class in Chapter 2; the FontMetrics class in Chapter 7

## getGraphics()

**InterfaceName**

ComponentPeer

**Purpose**

Obtains information about the graphics context of the Component that created this peer object.

**Syntax**

public abstract Graphics getGraphics()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to get the graphics context information for the Component.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

The Graphics object returned by this method describes the graphics context of the component. A value of null is returned if the Component is not currently displayed on the screen.

**See Also**

The getGraphics method of the Component class described in Chapter 2; the Graphics class described in Chapter 1

## getToolkit()

**InterfaceName**

ComponentPeer

**Purpose**

Gets an instance of the GUI-dependent toolkit used to bind the AWT classes to a particular graphics environment implementation.

**Syntax**

public abstract Toolkit getToolkit()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to return a Toolkit object for the underlying graphics environment.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

The Toolkit object returned by this method is a GUI-specific implementation of the Toolkit class. It is the interface between the AWT graphical components and the graphical components of the native graphical toolkit.

**See Also**

The getToolkit method of the Component class described in Chapter 2; the Toolkit class described in Chapter 3

## handleEvent(Event)

**InterfaceName**

ComponentPeer

**Purpose**

The event handler for events that occur in the Component object that are neither handled by the object's event handling method nor its container's event handling method.

**Syntax**

public abstract boolean handleEvent(Event e)

**Parameters**

***e***

The event that was detected.

**Description**

The peer object's handleEvent method is invoked to handle events that are neither handled by the component's handleEvent method nor by its container's handleEvent method. If a component's event handling method (handleEvent) does not handle an event, the event is passed to the event handling method of the component's container. If this container does not handle the event either, then the event is passed to this method of the component's peer object.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

This method returns true if it handles the event, and false if it does not handle the event.

**See Also**

The handleEvent method of the Component class described in Chapter 2; the Event class described in Chapter 3

## hide()

**InterfaceName**
ComponentPeer
**Purpose**
Hides the Component so that it is not visible on the screen.
**Syntax**
public abstract void hide()
**Parameters**
None.
**Description**
The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to hide the component.
**Imports**
*import java.awt.peer.ComponentPeer;*
**Returns**
None.
**See Also**
The hide method of the Component class described in Chapter 2

## minimumSize()

**InterfaceName**
ComponentPeer
**Purpose**
Determines the minimum dimensions of the Component object.
**Syntax**
public abstract Dimension minimumSize()
**Parameters**
None.
**Description**
The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to determine the minimum dimensions of the Component associated with this peer object.
**Imports**
*import java.awt.peer.ComponentPeer;*
**Returns**
This method returns a Dimension object that contains the minimum width and height measurements required of the Component object.
**See Also**
The minimumSize method of the Component class described in Chapter 2

## nextFocus()

**InterfaceName**
ComponentPeer
**Purpose**
Moves the input focus to the next component.

**Syntax**

public abstract void nextFocus()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to shift the input focus to the next component in the GUI.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The nextFocus method of the Component class described in Chapter 2

## paint(Graphics)

**InterfaceName**

ComponentPeer

**Purpose**

Paints the Component on the output device using the specified graphics context.

**Syntax**

public abstract void paint(Graphics g)

**Parameters**

***g***

The graphics context object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment in order to paint the appearance of the Component onto the output device whose characteristics are specified by the graphics context object *g*.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The paint method of the Component class described in Chapter 2

## preferredSize()

**InterfaceName**

ComponentPeer

**Purpose**

Determines the ideal dimensions of the Component object.

**Syntax**

public abstract Dimension preferredSize()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to determine the preferred dimensions of the Component object.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

This method returns a Dimension object that contains the ideal width and height measurements for the Component object.

**See Also**

The preferredSize method of the Component class described in Chapter 2

## prepareImage(Image, int, int, ImageObserver)

**InterfaceName**

ComponentPeer

**Purpose**

Prepares an image (asynchronously) for rendering onto the current output device.

**Syntax**

public abstract boolean prepareImage(Image img, int w, int h, ImageObserver o)

**Parameters**

***img***

The image to prepare for rendering onto this Component.

***w***

The width of the image to be prepared for rendering.

***h***

The height of the image to be prepared for rendering.

***o***

This parameter denotes the ImageObserver object that is notified as the image specified by *img* is being prepared for rendering.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to prepare an image for rendering by creating a scaled representation of it.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

This method returns the value of true if the image to be rendered is already prepared and ready for rendering, and false if it is not prepared.

**See Also**

The prepareImage method of the Component class described in Chapter 2

## print(Graphics)

**InterfaceName**

ComponentPeer

**Purpose**

Prints the Component on the device whose graphics context is specified.

**Syntax**

public abstract void print(Graphics g)

**Parameters**

***g***

The graphics context object to use for printing this Component.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to print the Component on the specified graphics context. The default implementation of this method is to invoke the paint method.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The print method of the Component class described in Chapter 2

## repaint(long, int, int, int, int)

**InterfaceName**

ComponentPeer

**Purpose**

Repaints the specified rectangular area of the Component and updates this area as soon as possible.

**Syntax**

public abstract void repaint(long tm, int x, int y, int width, int height)

**Parameters**

***tm***

The maximum amount of time (expressed in milliseconds) before the update method is invoked.

***x***

The x coordinate of the top-left part of the area of the Component to be repainted.

***y***

The y coordinate of the top-left part of the area of the Component to be repainted.

***width***

The width of the area of the Component to be repainted.

***height***

The height of the area of the Component to be repainted.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to repaint and update the rectangular area defined by the parameters to this method.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The repaint method of the Component class described in Chapter 2

## requestFocus()

**InterfaceName**

ComponentPeer

**Purpose**

Requests that the Component associated with this peer receive the input focus.

**Syntax**

public abstract void requestFocus()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment requesting the native toolkit to change the input focus to the Component associated with this peer object.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The requestFocus method of the Component class described in Chapter 2

## reshape(int, int, int, int)

**InterfaceName**

ComponentPeer

**Purpose**

Changes the shape of the Component such that it fits into the specified rectangular area.

**Syntax**

public abstract void reshape(int x, int y, int width, int height)

**Parameters**

***x***

The x coordinate of the top-left corner of the bounding box.

***y***

The y coordinate of the top-left corner of the bounding box.

***width***

The width of the bounding box.

***height***

The height of the bounding box.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to alter the shape of the Component such that it fits the specified bounding box.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The reshape method of the Component class described in Chapter 2

## setBackground(Color)

**InterfaceName**

ComponentPeer

**Purpose**

Sets the background color of the Component to the specified color.

**Syntax**

public abstract void setBackground(Color c)

**Parameters**

***c***

The Color value for the background of the Component associated with this peer object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to change the color used for painting the background of the Component to the specified color.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The setBackground method of the Component class described in Chapter 2

## setFont(Font)

**InterfaceName**

ComponentPeer

**Purpose**

Changes the font used in the Component to the specified font.

**Syntax**

public abstract void setFont(Font f)

**Parameters**

***f***

The font for the Component associated with this peer object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to change the font used for writing text in the Component.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The setFont method of the Component class described in Chapter 2

## setForeground(Color)

**InterfaceName**

ComponentPeer

**Purpose**

Sets the foreground color of the Component to the specified color

**Syntax**

public abstract void setForeground(Color c)

**Parameters**

***c***

The foreground color of the Component associated with this peer object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to change the color used for painting the foreground of the Component to the specified color.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The setForeground method of the Component class described in Chapter 2

## show()

**InterfaceName**

ComponentPeer

**Purpose**

Displays the Component.

**Syntax**

public abstract void show()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to display the Component on the current output device.

**Imports**

*import java.awt.peer.ComponentPeer;*

**Returns**

None.

**See Also**

The show method of the Component class described in Chapter 2.

## ButtonPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a button object.

**Syntax**

public interface ButtonPeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the Button class and the underlying operating system GUI primitives used with button objects. Native code that uses the API of the underlying GUI toolkit to create and manage a button object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-4 shows the inheritance hierarchy for the ButtonPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.ButtonPeer;*

**Constructors**

None.

**Parameters**

None.

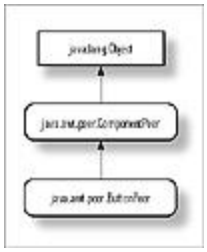

**Figure 9-4** Inheritance hierarchy for the ButtonPeer interface

## setLabel(String)

**InterfaceName**

ButtonPeer

**Purpose**

Displays the specified string on the button.

**Syntax**

public abstract void setLabel(String *label*)

**Parameters**

***label***

The text string for the Button label.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to set the label of the button object to the string specified as the parameter to this method.

**Imports**

*import java.awt.peer.ButtonPeer;*

**Returns**

None.

**See Also**

The setLabel method of the Button class described in Chapter 4

## CanvasPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a Canvas object.

**Syntax**

public interface CanvasPeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the Canvas class and the underlying operating system GUI primitives used with general-purpose Canvas objects. Native code that uses the API of the underlying GUI toolkit to create and manage a Canvas object is encapsulated in a GUI-dependent class that implements this interface. The GUI-dependent class that implements the ComponentPeer interface implements all the GUI-dependent functionality that is required of a Canvas object. This GUI-dependent class is the interface between the Canvas class and the underlying GUI toolkit of the platform on which the Java applications are being executed. Figure 9-5 shows the inheritance hierarchy for the CanvasPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.CanvasPeer;*

**Constructors**

None.

**Parameters**

None.

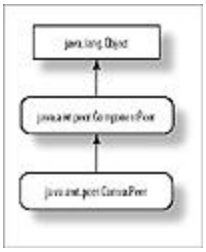

**Figure 9-5** Inheritance hierarchy for the CanvasPeer interface

## CheckboxPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a checkbox object.

**Syntax**

public interface CheckboxPeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the Checkbox class and the underlying operating system GUI primitives used with Checkbox objects. Native code that uses the API of the underlying GUI toolkit to create and manage a Checkbox object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-6 shows the inheritance hierarchy for the CheckboxPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.CheckboxPeer;*

**Constructors**

None.

**Parameters**

None.

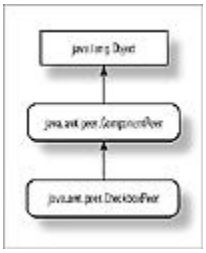

**Figure 9-6** Inheritance hierarchy for the CheckboxPeer interface

## setCheckboxGroup(CheckboxGroup)

**InterfaceName**

CheckboxPeer

**Purpose**

Associates the Checkbox object that created this peer object with the CheckboxGroup object specified

**Syntax**

public abstract void setCheckboxGroup(CheckboxGroup g)

**Parameters**

***g***

The CheckboxGroup object to which the Checkbox object associated with this CheckboxPeer object is assigned.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to associate the Checkbox object that created this peer object with the CheckboxGroup object specified as the parameter to this method.

**Imports**

*import java.awt.peer.CheckboxPeer;*

**Returns**

None.

**See Also**

The setCheckboxGroup method of the Checkbox class; the CheckboxGroup class described in Chapter 6

## setLabel(String)

**InterfaceName**

CheckboxPeer

**Purpose**

Sets the text label of the Checkbox

**Syntax**

public abstract void setLabel(String label)

**Parameters**

***label***

The text string to display on the label of the Checkbox.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to set the label of the Checkbox object associated with this peer object to the string specified as the parameter to this method.

**Imports**

*import java.awt.peer.CheckboxPeer;*

**Returns**

None.

**See Also**

The setLabel method of the Checkbox class described in Chapter 6

## setState(boolean)

**InterfaceName**

CheckboxPeer

**Purpose**

Sets the checkbox state to on or off.

**Syntax**

public abstract void setState(boolean state)

**Parameters**

***state***

A value of true causes the check mark to appear, and a value of false "unchecks" the Checkbox object associated with this CheckboxPeer object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to set the state of the Checkbox to either on (checked) or off (unchecked), depending on whether the value of *state* is true or false.

**Imports**

*import java.awt.peer.CheckboxPeer;*

**Returns**

None.

**See Also**

The setState method of the Checkbox class described in Chapter 6

## ChoicePeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a pop-up menu of choices.

**Syntax**

public interface ChoicePeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the Choice class and the underlying operating system GUI primitives used for creating and interacting with pop-up menu objects. Native code that uses the API of the underlying GUI toolkit to create and manage a pop-up menu of choices is encapsulated in a GUI-dependent class that implements this interface. Figure 9-7 shows the inheritance hierarchy for the ChoicePeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.ChoicePeer;*

**Constructors**

None.

**Parameters**

None.

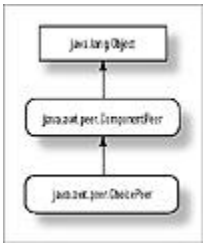

**Figure 9-7** Inheritance hierarchy for the ChoicePeer interface

## addItem(String, int)

**InterfaceName**
ChoicePeer
**Purpose**
Adds a text string to the pop-up menu at the specified position in the list of choices.
**Syntax**
public abstract void addItem(String item, int index)
**Parameters**
***item***
The text to insert into the Choice object associated with this ChoicePeer object.
***index***
The index at which to insert the specified text in the list of choices.
**Description**
The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to add the specified text string to the pop-up menu of choices.
**Imports**
*import java.awt.peer.ChoicePeer;*
**Returns**
None.
**See Also**
The addItem method of the Choice class described in Chapter 6

## select(int)

**InterfaceName**
ChoicePeer
**Purpose**
Selects the text string at the *index* position in the list of choices.
**Syntax**
public abstract void select(int index)
**Parameters**
***index***
The index into the list of choices of the item to select.
**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to select the specified item in the Choice object associated with this peer object.

**Imports**

*import java.awt.peer.ChoicePeer;*

**Returns**

None.

**See Also**

The select method of the Choice class described in Chapter 6

## LabelPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a label object.

**Syntax**

public interface LabelPeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the Label class and the underlying operating system GUI primitives used for creating and interacting with simple label objects that display a noneditable text string. Native code that uses the API of the underlying GUI toolkit to create and manage a label object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-8 shows the inheritance hierarchy for the LabelPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.LabelPeer*;

**Constructors**

None.

**Parameters**

None.

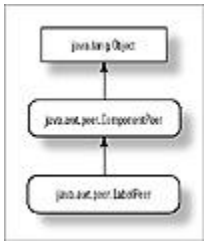

**Figure 9-8** Inheritance hierarchy for the LabelPeer interface

## setAlignment(int)

**InterfaceName**

LabelPeer

**Purpose**

Sets the alignment of the text string on the label.

**Syntax**

public abstract void setAlignment(int alignment)

**Parameters**

***alignment***

The alignment mode for the text string to be displayed in the Label object for which this LabelPeer object was created.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to align the label object's text string as specified by the *alignment* parameter.

**Imports**

*import java.awt.peer.LabelPeer;*

**Returns**

None.

**See Also**

The setAlignment method of the Label class described in Chapter 4

## setText(String)

**InterfaceName**

LabelPeer

**Purpose**

Changes the text string displayed on the label.

**Syntax**

public abstract void setText(String label)

**Parameters**

***label***

The text for the label.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to display the specified text on the label object.

**Imports**

*import java.awt.peer.LabelPeer;*

**Returns**

None.

**See Also**

The setText method of the Label class described in Chapter 4

## ListPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a list object.

**Syntax**

public interface ListPeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the List class and the underlying operating system GUI primitives used for creating and interacting with graphical objects that contain a scrolling list of text strings. Native code that uses the API of the underlying GUI toolkit to create and manage a list object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-9 shows the inheritance hierarchy for the ListPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.ListPeer;*

**Constructors**

None.

**Parameters**

None.

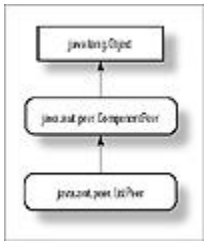

**Figure 9-9** Inheritance hierarchy for the ListPeer interface

## addItem(String, int)

**InterfaceName**

ListPeer

**Purpose**

Adds a text string at a specified position in the list.

**Syntax**

public abstract void addItem(String item, int index)

**Parameters**

***item***

The text string to add to the list.

***index***

The index at which to insert the specified text

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to add the specified string at the specified index in the list object for which this peer object was created.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

None.

**See Also**

The addItem method of the List class described in Chapter 5

## clear()

**InterfaceName**

ListPeer

**Purpose**

Empties the list.

**Syntax**

public abstract void clear()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to remove all the text strings from the list object.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

None.

**See Also**

The clear method of the List class described in Chapter 5

## delItems(int, int)

**InterfaceName**

ListPeer

**Purpose**

Removes a specified range of items from the list object.

**Syntax**

public abstract void delItems(int start, int end)

**Parameters**

***start***

The position of the first item to be deleted.

***end***

The index of the last item to be removed.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to remove all the text strings between the *start* and *end* indices in the list.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

None.

**See Also**

The delItems method of the List class described in Chapter 5

## deselect(int)

**InterfaceName**

ListPeer

**Purpose**

Deselects the item at the specified index in the list.

**Syntax**

public abstract void deselect(int index)

**Parameters**

***index***

The position of the text string to deselect.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to remove the highlight bar from around the item at the position specified by *index*.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

None.

**See Also**

The deselect method of the List class described in Chapter 5

## getSelectedIndexes()

**InterfaceName**

ListPeer

**Purpose**

Gets the indices of the selected items in the list.

**Syntax**

public abstract int[] getSelectedIndexes()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the items in the list that have been selected.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

This method returns an array containing integer values that indicate the positions that have been selected in the list.

**See Also**

The getSelectedIndexes method of the List class described in Chapter 5

## makeVisible(int)

**InterfaceName**

ListPeer

**Purpose**

Forces the item at the specified index in the list to be made visible.

**Syntax**

public abstract void makeVisible(int index)

**Parameters**

***index***

The index of the item in the list that is forced to be made visible.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to bring the specified text item into the visible area of the list box, scrolling the items in the list object if necessary.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

None.

**See Also**

The makeVisible method of the List class described in Chapter 5

## minimumSize()

**InterfaceName**

ListPeer

**Purpose**

Determines the minimum height and width for a List object containing the specified number of rows.

**Syntax**

public abstract Dimension minimumSize(int v)

**Parameters**

***v***

The number of rows in the list.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the minimum dimension required if the List object associated with this peer object has *v* number of rows.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

This method returns a Dimension object that contains the minimum width and height measurements required of the List object.

**See Also**

The minimumSize method of the List class described in Chapter 5

## preferredSize(int)

**InterfaceName**

ListPeer

**Purpose**

Determines the ideal dimensions for a List object containing the specified number of rows.

**Syntax**

public abstract Dimension preferredSize(int v)

**Parameters**

***v***

The number of rows in the list.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the ideal dimension required if the List object associated with this peer object has *v* number of rows.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

This method returns a Dimension object that contains the ideal width and height measurements required of the List object.

**See Also**

The preferredSize method of the List class described in Chapter 5

## select(int)

**InterfaceName**

ListPeer

**Purpose**

Selects the text item at the specified position in the list.

**Syntax**

public abstract void select(int index)

**Parameters**

***index***

The index of the item in the list to be selected.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to highlight the item at the position specified by *index*.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

None.

**See Also**

The select method of the List class described in Chapter 5

## setMultipleSelections(boolean)

**InterfaceName**

ListPeer

**Purpose**

Enables and disables multiple selection of items in the list.

**Syntax**

public abstract void setMultipleSelections(boolean mFlag)

**Parameters**

***mFlag***

If the value of *mFlag* is true, then multiple items in the list can be selected. If the value of *mFlag* is false, then the list functions as a single-selection list in which only one of the items displayed can be selected.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI, to set whether or not the user can select multiple items in the List component associated with this peer object.

**Imports**

*import java.awt.peer.ListPeer;*

**Returns**

None.

**See Also**

The setMultipleSelections method of the List class described in Chapter 5

## ScrollbarPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a scrollbar object.

**Syntax**

public interface ScrollbarPeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the Scrollbar class and the underlying operating system GUI primitives used for creating and interacting with scrollbar objects. Native code that uses the API of the underlying GUI toolkit to create and manage a scrollbar object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-10 shows the inheritance hierarchy for the ScrollbarPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.ScrollbarPeer;*

**Constructors**

None.

**Parameters**

None.

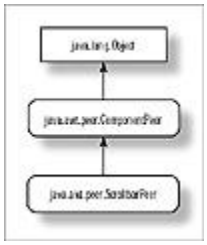

**Figure 9-10** Inheritance hierarchy for the ScrollbarPeer interface

## setLineIncrement(int)

**InterfaceName**

ScrollbarPeer

**Purpose**

Sets the step size for decrements and increments when the line up or line down arrow buttons of the scrollbar are invoked.

**Syntax**

public abstract void setLineIncrement(int l)

**Parameters**

***l***

The line increment size.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set the step size for line increments and decrements of the Scrollbar object associated with this peer object.

**Imports**

*import java.awt.peer.ScrollbarPeer;*

**Returns**

None.

**See Also**

The setLineIncrement method of the Scrollbar class described in Chapter 4

## setPageIncrement(int)

**InterfaceName**

ScrollbarPeer

**Purpose**

Sets the step size for decrements or increments when the page up or page down arrow buttons of the scrollbar are invoked.

**Syntax**

public abstract void setPageIncrement(int l)

**Parameters**

***l***

The page increment size.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set the step size for page increments and decrements of the Scrollbar object associated with this peer object.

**Imports**

*import java.awt.peer.ScrollbarPeer;*

**Returns**

None.

**See Also**

The setPageIncrement method of the Scrollbar class described in Chapter 4

## setValue(int)

**InterfaceName**

ScrollbarPeer

**Purpose**

Sets the value of the current position of the Scrollbar to the specified value.

**Syntax**

public abstract void setValue(int p)

**Parameters**

***p***

The new value for the current position of the Scrollbar.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set the value of the current position of the Scrollbar object associated with this peer object.

**Imports**

*import java.awt.peer.ScrollbarPeer;*

**Returns**

None.

**See Also**

The setValue method of the Scrollbar class described in Chapter 4

## setValues(int, int, int, int)

**InterfaceName**

ScrollbarPeer

**Purpose**

Sets various parameters associated with the ScrollBar object.

**Syntax**

public abstract void setValues(int value, int visible, int minScroll, int maxScroll)

**Parameters**

***value***

The position of the scrollbar thumb in the current window.

***visible***

The size of the visible region of the area that is being scrolled using the scrollbar.

***minScroll***

The minimum value of the scrollbar.

***maxScroll***

The maximum value of the scrollbar.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set the specified parameters for the Scrollbar object associated with this peer object.

**Imports**

*import java.awt.peer.ScrollbarPeer;*

**Returns**

None

**See Also**

The setValues method of the Scrollbar class described in Chapter 4

## ContainerPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a container object.

**Syntax**

public interface ContainerPeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the Container class and the underlying operating system GUI primitives used for creating and interacting with graphical components that can contain other graphical components. Native code that uses the API of the underlying GUI toolkit to create and manage a GUI component that can contain other graphical components is encapsulated in a GUI-dependent class that implements this interface. Figure 9-11 shows the inheritance hierarchy for the ContainerPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.ContainerPeer;*

**Constructors**

None.

**Parameters**

None.

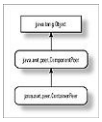

**Figure 9-11** Inheritance hierarchy for the ContainerPeer interface

## insets()

**InterfaceName**
ContainerPeer
**Purpose**
Determines the top, left, bottom, right insets for the Container object.
**Syntax**
public abstract Insets insets()
**Parameters**
None.
**Description**
The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the extra space that must be added as padding on the top, left, bottom, and right sides of the Container object.
**Imports**
*import java.awt.peer.ContainerPeer;*
**Returns**
This method returns an Insets object that specifies the top, left, bottom, and right padding that must be subtracted from the dimensions of the Container object in order to determine the area required for laying out Components within the Container.
**See Also**
The insets method of the Container class described in Chapter 3

## PanelPeer

**Purpose**
Defines the interface that must be implemented by a GUI-dependent class to create and manage a panel object.
**Syntax**
public interface PanelPeer extends Object extends ContainerPeer
**Description**
This interface defines the API between the Panel class and the underlying operating system GUI primitives used for creating and interacting with panel graphical components. Native code that uses the API of the underlying GUI toolkit to create and manage a panel object is encapsulated in a GUI-dependent class that implements this interface. This GUI-dependent class is the interface between the *Panel* class and the underlying GUI toolkit of the platform on which the Java applications are being executed. Figure 9-12 shows the inheritance hierarchy for the PanelPeer interface.
**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.PanelPeer;*

**Constructors**

None.

**Parameters**

None.

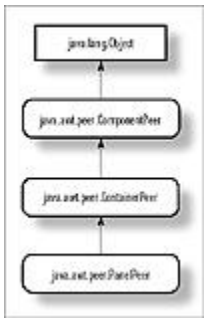

**Figure 9-12** Inheritance hierarchy for the PanelPeer interface

## WindowPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a top-level window object.

**Syntax**

public interface WindowPeer extends Object extends ContainerPeer

**Description**

This interface defines the API between the Window class and the underlying operating system GUI primitives used for creating and interacting with top-level windows. These windows have neither a title bar nor a border. Native code that uses the API of the underlying GUI toolkit to create and manage a top-level container object, is encapsulated in a GUI-dependent class that implements this interface. Figure 9-13 shows the inheritance hierarchy for the WindowPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.WindowPeer;*

**Constructors**

None.

**Parameters**

None.

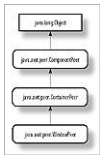

**Figure 9-13** Inheritance hierarchy for the WindowPeer interface

## toBack()

**InterfaceName**
WindowPeer
**Purpose**
Sends the parent frame object to the back of the Window object associated with this peer.
**Syntax**
public abstract void toBack()
**Parameters**
None.
**Description**
The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to push the parent Frame window object to the back of the Window.
**Imports**
*import java.awt.peer.WindowPeer;*
**Returns**
None.
**See Also**
The toBack method of the Window class described in Chapter 3

## toFront()

**InterfaceName**
WindowPeer
**Purpose**
Brings the parent frame object to the front of the Window object associated with this peer.
**Syntax**
public abstract void toFront()
**Parameters**
None.
**Description**
The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to bring the parent Frame window object to the front of the Window object.
**Imports**
*import java.awt.peer.WindowPeer;*
**Returns**

None.

**See Also**

The toFront method of the Window class described in Chapter 3

## DialogPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a Dialog box object.

**Syntax**

public interface DialogPeer extends Object extends WindowPeer

**Description**

This interface defines the API between the Dialog class and the underlying operating system GUI primitives used for creating and interacting with dialog box object. The GUI-dependent class that creates a dialog box object must implement this interface. The methods defined in this interface are implemented in native code, using the API of the underlying GUI toolkit to create and manage a dialog box object. Figure 9-14 shows the inheritance hierarchy for the DialogPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.DialogPeer;*

**Constructors**

None.

**Parameters**

None.

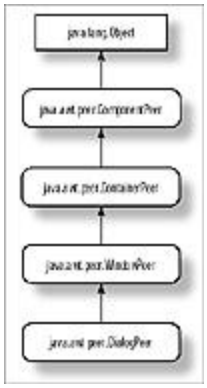

**Figure 9-14** Inheritance hierarchy for the DialogPeer interface

## setResizable(boolean)

**InterfaceName**

DialogPeer

**Purpose**

Sets whether the dialog box can be resized or not.

**Syntax**

public abstract void setResizable(boolean resizable)

**Parameters**

***resizable***

A value of true makes the Dialog component resizable; false means the Dialog window is not resizable.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to enable and disable the resizable property of the Dialog window object associated with this peer.

**Imports**

*import java.awt.peer.DialogPeer;*

**Returns**

None.

**See Also**

The setResizable method of the Dialog class described in Chapter 5

## setTitle(String)

**InterfaceName**

DialogPeer

**Purpose**

Sets the text string on the title bar of the Dialog window.

**Syntax**

public abstract void setTitle(String title)

**Parameters**

***title***

The text to display in the title bar of the Dialog window.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set the title of the Dialog window to the string specified in the parameter to this method.

**Imports**

*import java.awt.peer.DialogPeer;*

**Returns**

None.

**See Also**

The setTitle method of the Dialog class described in Chapter 5

## FileDialogPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a FileDialog window that displays a list of files and allows the user to select a file.

**Syntax**

public interface FileDialogPeer extends Object extends DialogPeer

**Description**

This interface defines the API between the FileDialog class and the underlying operating system GUI primitives used for creating and interacting with dialog box windows that display a list of files and allow the user to select from the list. The GUI-dependent class that creates a file dialog box window must implement this interface. The methods defined in this interface are implemented in native code using the API of the underlying GUI toolkit to create and manage a dialog box window that allows the user to pick a file from a list of filenames. Figure 9-15 shows the inheritance hierarchy for the FileDialogPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.FileDialogPeer;*

**Constructors**

None.

**Parameters**

None.

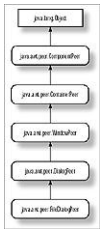

**Figure 9-15** Inheritance hierarchy for the FileDialogPeer interface

## setDirectory(String)

**InterfaceName**

FileDialogPeer

**Purpose**

Sets the directory from which the user is prompted to select a file.

**Syntax**

public abstract void setDirectory(String dir)

**Parameters**

***dir***

The name of the directory from which the user is prompted to select a file.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to display the text string specified in *dir* in a window on the FileDialog box.

**Imports**

*import java.awt.peer.FileDialogPeer;*

**Returns**

None.

**See Also**

The setDirectory method of the FileDialog class described in Chapter 5

## setFile(String)

**InterfaceName**

FileDialogPeer

**Purpose**

Sets the directory from which the user is prompted to select a file.

**Syntax**

public abstract void setFile(String file)

**Parameters**

***file***

The name of the (default) file for the FileDialog object associated with this peer object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to display the text string specified in *file* as a label on the FileDialog box.

**Imports**

*import java.awt.peer.FileDialogPeer;*

**Returns**

None.

**See Also**

The setFile method of the FileDialog class described in Chapter 5

## setFilenameFilter(FilenameFilter)

**InterfaceName**

FileDialogPeer

**Purpose**

Sets the filter that determines which files to display for users to select from in the FileDialog box.

**Syntax**

public abstract void setFilenameFilter(FilenameFilter filter)

**Parameters**

***filter***

The file name filter for displaying files in the FileDialog window.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to display the text string specified in *filter*

on a label on the FileDialog box and to perform the appropriate filtering of file names.

**Imports**

*import java.awt.peer.FileDialogPeer;*

**Returns**

None.

**See Also**

The setFilenameFilter method of the FileDialog class described in Chapter 5

## FramePeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a Frame window.

**Syntax**

public interface FramePeer extends Object extends WindowPeer

**Description**

This interface defines the API between the Frame class and the underlying operating system GUI primitives used for creating and interacting with top-level frame windows that have a title bar and borders and can optionally have a menu bar. Native code that uses the API of the underlying GUI toolkit to create and manage a top-level frame window object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-16 shows the inheritance hierarchy for the FramePeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.FramePeer;*

**Constructors**

None.

**Parameters**

None.

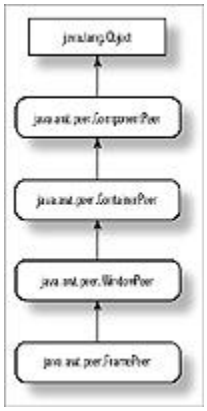

**Figure 9-16** Inheritance hierarchy for the FramePeer interface

## setCursor(int)

**InterfaceName**

FramePeer

**Purpose**

Sets the cursor to display when the pointer is within the Frame window associated with this peer object.

**Syntax**

public abstract void setCursor(int cursorType)

**Parameters**

***cursorType***

An integer constant that indicates the type of cursor to display within the Frame window for which this FramePeer was created.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to display the appropriate bitmap image when the pointer is within the bounds of the Frame window associated with this peer.

**Imports**

*import java.awt.peer.FramePeer;*

**Returns**

None.

**See Also**

The setCursor method of the Frame class described in Chapter 4

## setIconImage(Image)

**InterfaceName**

FramePeer

**Purpose**

Sets the image of the icon of the Frame window associated with this peer object.

**Syntax**

public abstract void setIconImage(Image img)

**Parameters**

***img***

The image to be used for the icon.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to display the appropriate image when the Frame window is iconized.

**Imports**

*import java.awt.peer.FramePeer;*

**Returns**

None.

**See Also**

The setIconImage method of the Frame class described in Chapter 4

## setMenuBar(MenuBar)

**InterfaceName**

FramePeer

**Purpose**

Sets the menu bar for the Frame to the specified MenuBar object.

**Syntax**

public abstract void setMenuBar(MenuBar mbar)

**Parameters**

***mbar***

The MenuBar object that represents the menu bar for the Frame window.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to associate the specified menu bar with the Frame window associated with this peer object.

**Imports**

*import java.awt.peer.FramePeer;*

**Returns**

None.

**See Also**

The setMenuBar method of the Frame class described in Chapter 4

## setResizable(boolean)

**InterfaceName**

FramePeer

**Purpose**

Sets whether or not the Frame window can be resized.

**Syntax**

public abstract void setResizable(boolean resizable)

**Parameters**

***resizable***

A boolean value that represents whether or not the Frame window is resizable. If this value is set to true, the Frame window is resizable; if it is set to false, the Frame window is not resizable.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to enable and disable the resizable property of the Frame window associated with this peer.

**Imports**

*import java.awt.peer.FramePeer;*

**Returns**

None.

**See Also**

The setResizable method of the Frame class described in Chapter 4

## setTitle(String)

**InterfaceName**

FramePeer

**Purpose**

Sets the text string on the title bar of the frame.

**Syntax**

public abstract void setTitle(String title)

**Parameters**

***title***

The text string to display in the title bar of the Frame window.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set the title of the Frame window to the string specified in the parameter to this method.

**Imports**

*import java.awt.peer.FramePeer;*

**Returns**

None.

**See Also**

The setTitle method of the Frame class described in Chapter 4

## MenuComponentPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a MenuComponent object.

**Syntax**

public interface MenuComponentPeer extends Object

**Description**

This interface defines the API between the MenuComponent class and the underlying operating system GUI primitives used for creating and interacting with graphical objects associated with menus. Native code that uses the API of the underlying GUI toolkit to create and manage a menu component object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-17 shows the inheritance hierarchy for the MenuComponentPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.MenuComponentPeer;*

**Constructors**

None.

**Parameters**

None.

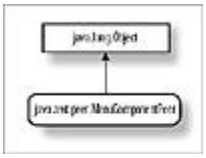

**Figure 9-17** Inheritance hierarchy for the MenuComponentPeer interface

## dispose()

**InterfaceName**
MenuComponentPeer
**Purpose**
Frees the resources allocated to the MenuComponent.
**Syntax**
public abstract void dispose()
**Parameters**
None.
**Description**
The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to free the resources allocated to the MenuComponent object associated with this peer.
**Imports**
*import java.awt.peer.MenuComponentPeer;*
**Returns**
None.
**See Also**
The removeNotify method of the MenuComponent class described in Chapter 6

## MenuBarPeer

**Purpose**
Defines the interface that must be implemented by a GUI-dependent class to create and manage a MenuBar object.
**Syntax**
public interface MenuBarPeer extends MenuComponentPeer
**Description**
This interface defines the API between the MenuBar class and the underlying operating system GUI primitives used for creating and interacting with menu bar objects. Native code that uses the API of the underlying GUI toolkit to create and manage a menu bar object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-18 shows the inheritance hierarchy for the MenuBarPeer interface.
**PackageName**
*java.awt.peer*
**Imports**

*import java.awt.peer.MenuBarPeer;*

**Constructors**

None.

**Parameters**

None.

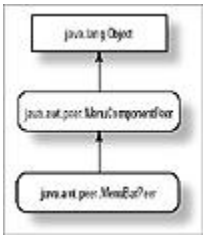

**Figure 9-18** Inheritance hierarchy for the MenuBarPeer interface

## addHelpMenu(Menu)

**InterfaceName**

MenuBarPeer

**Purpose**

Adds the specified menu as the help menu for the menu bar.

**Syntax**

public abstract void addHelpMenu(Menu m)

**Parameters**

***m***

The help menu on the menu bar is set to this menu object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to add the specified menu to the menu bar and to designate it as the help menu.

**Imports**

*import java.awt.peer.MenuBarPeer;*

**Returns**

None.

**See Also**

The setHelpMenu method of the MenuBar class described in Chapter 6

## addMenu(Menu)

**InterfaceName**

MenuBarPeer

**Purpose**

Adds a menu to the MenuBar object associated with this peer object.

**Syntax**

public abstract void addMenu(Menu m)

**Parameters**

***m***

The menu object to be added to the menu bar.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to add the specified menu to the menu bar.

**Imports**

*import java.awt.peer.MenuBarPeer;*

**Returns**

None.

**See Also**

The add method of the MenuBar class described in Chapter 6

## delMenu(int)

**InterfaceName**

MenuBarPeer

**Purpose**

Removes the specified menu from the menu bar.

**Syntax**

public abstract void delMenu(int index)

**Parameters**

***index***

The index of the menu to remove from the menu bar object associated with this peer.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to remove the specified menu from the menu bar.

**Imports**

*import java.awt.peer.MenuBarPeer;*

**Returns**

None.

**See Also**

The remove method of the MenuBar class described in Chapter 6

## MenuItemPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a MenuItem object.

**Syntax**

public interface MenuItemPeer extends MenuComponentPeer

**Description**

This interface defines the API between the MenuItem class and the underlying operating system GUI primitives used for creating and interacting with menu item

objects. Native code that uses the API of the underlying GUI toolkit to create and manage a menu item object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-19 shows the inheritance hierarchy for the MenuItemPeer interface.

**PackageName**

*import java.awt.peer*

**Imports**

*import java.awt.peer.MenuBarPeer;*

**Constructors**

None.

**Parameters**

None.

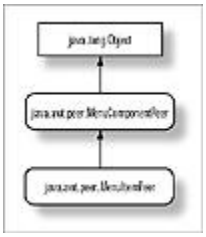

**Figure 9-19** Inheritance hierarchy for the MenuItemPeer interface

## disable()

**InterfaceName**

MenuItemPeer

**Purpose**

Makes the MenuItem associated with this peer unselectable by the user.

**Syntax**

public abstract void disable()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to make the menu item unselectable by the user.

**Imports**

*import java.awt.peer.MenuItemPeer;*

**Returns**

None.

**See Also**

The disable method of the MenuItem class described in Chapter 6

## enable()

**InterfaceName**

MenuItemPeer

**Purpose**

Makes the MenuItem associated with this peer selectable by the user.

**Syntax**

public abstract void enable()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to make the menu item selectable by the user.

**Imports**

*import java.awt.peer.MenuItemPeer;*

**Returns**

None.

**See Also**

The enable method of the MenuItem class described in Chapter 6

## setLabel(String)

**InterfaceName**

MenuItemPeer

**Purpose**

Sets the text label of the MenuItem to the specified string.

**Syntax**

public abstract void setLabel(String label)

**Parameters**

***label***

The text for the label of the MenuItem for which this peer was created.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set the text of the menu item to the string specified in *label*.

**Imports**

*import java.awt.peer.MenuItemPeer;*

**Returns**

None.

**See Also**

The setLabel method of the MenuItem class described in Chapter 6

## CheckboxMenuItemPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a CheckboxMenuItem object.

**Syntax**

public interface CheckboxMenuItemPeer extends Object extends MenuItemPeer

**Description**

This interface defines the API between the CheckboxMenuItem class and the underlying operating system GUI primitives used for creating and interacting with checkboxes that can be used as menu item objects. Native code that uses the API of the underlying GUI toolkit to create and manage a checkbox object that can be used in menus is encapsulated in a GUI-dependent class that implements this interface. Figure 9-20 shows the inheritance hierarchy for the CheckboxMenuItemPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.CheckboxMenuItemPeer;*

**Constructors**

None.

**Parameters**

None.

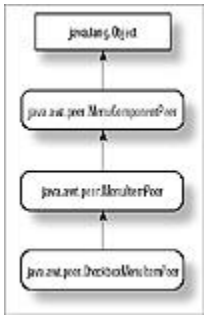

**Figure 9-20** Inheritance hierarchy for the CheckboxMenuItemPeer interface

## setState(boolean)

**InterfaceName**

CheckboxMenuItemPeer

**Purpose**

Sets the checkbox state (in the menu) to either on or off.

**Syntax**

public abstract void setState(boolean t)

**Parameters**

***t***

The checkbox is “unchecked” if this value is false. The checkbox is “checked” if the value of this variable is true.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific graphical environment to set the state of the Checkbox to either on (checked) or off (unchecked), depending on whether the value of *t* is true or false.

**Imports**

*import java.awt.peer.CheckboxMenuItemPeer;*

**Returns**

None.

**See Also**

The setState method of the CheckboxMenuItem class described in Chapter 6

## MenuPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a Menu object.

**Syntax**

public interface MenuPeer extends MenuItemPeer

**Description**

This interface defines the API between the Menu class and the underlying GUI primitives that can be components of a menu bar. Native code that uses the API of the underlying GUI toolkit to create and manage a menu object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-21 shows the inheritance hierarchy for the MenuPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.MenuPeer;*

**Constructors**

None.

**Parameters**

None.

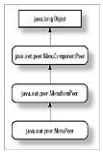

**Figure 9-21** Inheritance hierarchy for the MenuPeer interface

## addItem(MenuItem)

**InterfaceName**

MenuPeer

**Purpose**

Adds the specified MenuItem to the Menu object associated with this peer.

**Syntax**

public abstract void addItem(MenuItem mItem)

**Parameters**

***mItem***

The MenuItem to be added to the menu.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to add the specified menu item to the Menu associated with this peer.

**Imports**

*import java.awt.peer.MenuPeer;*

**Returns**

None.

**See Also**

The add method of the Menu class described in Chapter 6

## addSeparator()

**InterfaceName**

MenuPeer

**Purpose**

Adds a separator line in the Menu.

**Syntax**

public abstract void addSeparator()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to draw a line just below the previous item that was added to the Menu object assoicated with this peer object.

**Imports**

*import java.awt.peer.MenuPeer;*

**Returns**

None.

**See Also**

The addSeparator method of the Menu class described in Chapter 6

## delItem(int)

**InterfaceName**

MenuPeer

**Purpose**

Removes the menu item at the index specified by *index* from the Menu object associated with this peer.

**Syntax**

public abstract void delItem(int index)

**Parameters**

***index***

The index of the item to remove from the Menu object associated with this peer.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to remove the MenuItem at the specified index from the Menu object.

**Imports**

*import java.awt.peer.MenuPeer;*

**Returns**

None.

**See Also**

The remove method of the Menu class described in Chapter 6

## TextComponentPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a TextComponent object.

**Syntax**

public interface TextComponentPeer extends Object extends ComponentPeer

**Description**

This interface defines the API between the TextComponent class and the underlying operating system GUI primitives used for creating graphical components in which text can be edited. Native code that uses the API of the underlying GUI toolkit to create and manage a TextComponent object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-22 shows the inheritance hierarchy for the TextComponentPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.TextComponentPeer;*

**Constructors**

None.

**Parameters**

None.

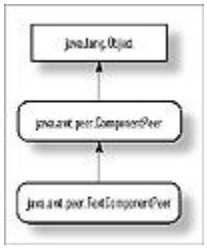

**Figure 9-22** Inheritance hierarchy for the TextComponentPeer interface

## getSelectionEnd()

**InterfaceName**

TextComponentPeer

**Purpose**

Returns the position of the end of the text that the user selected.

**Syntax**

public abstract int getSelectionEnd()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the position of the end of the text selected by the user.

**Imports**

*import java.awt.peer.TextComponentPeer;*

**Returns**

This method returns an integer value that indicates the character position of the last character in the text selected by the user.

**See Also**

The getSelectionEnd method of the TextComponent class described in Chapter 5

## getSelectionStart()

**InterfaceName**

TextComponentPeer

**Purpose**

Returns the position of the start of the text that the user selected.

**Syntax**

public abstract int getSelectionStart()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the position of the beginning of the text selected by the user.

**Imports**

*import java.awt.peer.TextComponentPeer;*

**Returns**

This method returns an integer value that indicates the character position of the first character in the text selected by the user.

**See Also**

The getSelectionStart method of the TextComponent class described in Chapter 5

## getText()

**InterfaceName**

TextComponentPeer

**Purpose**

Returns the text that the TextComponent contains.

**Syntax**

public abstract String getText()

**Parameters**

None.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the text contained in the TextComponent object associated with this peer.

**Imports**

*import java.awt.peer.TextComponentPeer;*

**Returns**

This method returns a string containing all the text contained in the TextComponent.

**See Also**

The getText method of the TextComponent class described in Chapter 5

## select(int, int)

**InterfaceName**

TextComponentPeer

**Purpose**

Selects a range of text.

**Syntax**

public abstract void select(int selStart, int selEnd)

**Parameters**

***selStart***

The position at which to start selecting the text.

***selEnd***

The position at which to stop selecting the text, starting from *selStart*.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to select the range of text between the start index *selStart* and the end index *selEnd*. Highlighting the selected text is a standard convention followed in most graphical environments.

**Imports**

*import java.awt.peer.TextComponentPeer;*

**Returns**

None.

**See Also**

The select method of the TextComponent class described in Chapter 5

## setEditable(boolean)

**InterfaceName**

TextComponentPeer

**Purpose**

Sets whether or not the TextComponent permits editing of the text contained in it.

**Syntax**

public abstract void setEditable(boolean editable)

**Parameters**

***editable***

A value of true makes the text in the TextComponent editable by the user. A value of false means the text in the TextComponent cannot be edited.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set whether or not the text contained in the TextComponent can be edited.

**Imports**

*import java.awt.peer.TextComponentPeer;*

**Returns**

None.

**See Also**

The setEditable method of the TextComponent class described in Chapter 5

## setText(String)

**InterfaceName**

TextComponentPeer

**Purpose**

Sets the string to display in the TextComponent object associated with this peer.

**Syntax**

public abstract void setText(String s)

**Parameters**

***s***

The text to display in the TextComponent object associated with this peer object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to display the text string *s* in the TextComponent.

**Imports**

*import java.awt.peer.TextComponentPeer;*

**Returns**

None.

**See Also**

The setText method of the TextComponent class described in Chapter 5

## TextAreaPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a TextArea object.

**Syntax**

public interface TextAreaPeer extends Object extends TextComponentPeer

**Description**

This interface defines the API between the TextArea class and the underlying operating system GUI primitives used for displaying, editing, and managing multiline text areas. Native code that uses the API of the underlying GUI toolkit to create and manage a TextArea object is encapsulated in a GUI-dependent class that implements this interface. Figure 9-23 shows the inheritance hierarchy for the TextAreaPeer interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.TextAreaPeer;*

**Constructors**

None.

**Parameters**

None.

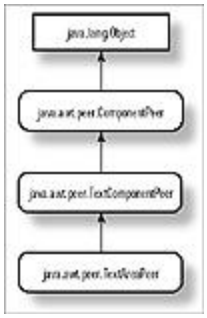

**Figure 9-23** Inheritance hierarchy for the TextAreaPeer interface

## insertText(String, int)

**InterfaceName**

TextAreaPeer

**Purpose**

Inserts a string of text at a specified index into the existing text in the TextArea object.

**Syntax**

public abstract void insertText(String txt, int pos)

**Parameters**

***txt***

The text to insert into the TextArea object.

***pos***

The position at which to insert the specified text.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to insert the text string(*txt*) at the index specified by *pos* into the TextArea component associated with this peer object.

**Imports**

*import java.awt.peer.TextAreaPeer;*

**Returns**

None.

**See Also**

The insertText method of the TextArea class described in Chapter 5

## minimumSize(int, int)

**InterfaceName**

TextAreaPeer

**Purpose**

Determines the minimum height and width for a TextArea object containing the specified number of rows and columns of text.

**Syntax**

public abstract Dimension minimumSize(int rows, int cols)

**Parameters**

***rows***

The number of rows in the TextArea object.

***cols***

The number of columns in the TextArea object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the minimum dimension required if the TextArea object associated with this peer object has the specified number of rows and columns of text.

**Imports**

*import java.awt.peer.TextAreaPeer;*

**Returns**

This method returns a Dimension object that contains the minimum width and height measurements required of the TextArea object.

**See Also**

The minimumSize method of the TextArea class described in Chapter 5

## preferredSize(int, int)

**InterfaceName**

TextAreaPeer

**Purpose**

Determines the ideal height and width for a TextArea object containing the specified number of rows and columns of text.

**Syntax**

public abstract Dimension preferredSize(int rows, int cols)

**Parameters**

***rows***

The preferred number of rows for the TextArea object.

***cols***

The preferred number of columns for the TextArea object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the ideal dimension required if the TextArea object associated with this peer object has the specified number of rows and columns of text.

**Imports**

*import java.awt.peer.TextAreaPeer;*

**Returns**

This method returns a Dimension object that contains the ideal width and height measurements required of the TextArea object.

**See Also**

The preferredSize method of the TextArea class described in Chapter 5

## replaceText(String, int, int)

**InterfaceName**

TextAreaPeer

**Purpose**

Replaces a specified range of text with another string of text.

**Syntax**

public abstract void replaceText (String txt, int start, int end)

**Parameters**

***txt***

The new text string to use as the replacement text.

***start***

The starting position of the text in the TextArea to be replaced.

***end***

The ending position of the text in the TextArea to be replaced.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to replace the characters between the indices specified by *start* and *end*, with the text string supplied in *txt*.

**Imports**

*import java.awt.peer.TextAreaPeer;*

**Returns**

None.

**See Also**

The replaceText method of the TextArea class described in Chapter 5

## TextFieldPeer

**Purpose**

Defines the interface that must be implemented by a GUI-dependent class to create and manage a TextField object.

**Syntax**

public interface TextFieldPeer extends Object extends TextComponentPeer

**Description**

This interface defines the API between the TextField class and the underlying operating system GUI primitives used for creating graphical components that allow editing of a single line of text. Native code that uses the API of the underlying GUI toolkit to create and manage a TextField object is encapsulated in a GUI-dependent class that implements this interface.

**PackageName**

*java.awt.peer*

**Imports**

*import java.awt.peer.TextFieldPeer;*

**Constructors**

None.

**Parameters**

None.

## minimumSize(int)

**InterfaceName**

TextFieldPeer

**Purpose**

Determines the minimum height and width for a TextField object containing the specified number of columns.

**Syntax**

public abstract Dimension minimumSize(int cols)

**Parameters**

***cols***

The minimum number of columns the TextField object must have.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the minimum dimension required if the TextField object associated with this peer object has *cols* number of columns of text.

**Imports**

*import java.awt.peer.TextFieldPeer;*

**Returns**

This method returns a Dimension object that contains the minimum width and height measurements required of the TextField object.

**See Also**

The minimumSize method of the TextField class described in Chapter 5

## preferredSize(int)

**InterfaceName**

TextFieldPeer

**Purpose**

Determines the ideal height and width for a TextField object containing the specified number of columns of text.

**Syntax**

public abstract Dimension preferredSize(int cols)

**Parameters**

***cols***

The preferred number of columns the TextField object must have.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to determine the ideal dimension of the TextField object associated with this peer object if it has *cols* number of columns.

**Imports**

*import java.awt.peer.TextFieldPeer;*

**Returns**

This method returns a Dimension object that contains the ideal width and height measurements required of the TextField object.

**See Also**

The preferredSize method of the TextField class described in Chapter 5

## setEchoCharacter(char)

**InterfaceName**

TextFieldPeer

**Purpose**

Sets the character that is echoed when the user enters text in the TextField object associated with this peer.

**Syntax**

public abstract void setEchoCharacter(char c)

**Parameters**

***c***

The character that should be printed in the TextField when the user enters text in the TextField object.

**Description**

The GUI-dependent class that implements this method must make native calls to the API of the platform-specific GUI to set the echo character for the TextField object to the specified character.

**Imports**

*import java.awt.peer.TextFieldPeer;*

**Returns**

None.

**See Also**

The setEchoCharacter method of the TextField class described in Chapter 5

# *PART IV*
# *Networking in Java*

# Chapter 10
# Network And Sockets

A computer network is a communication system for connecting two or more hosts. Hosts can be anything from microcomputers to super-computers, which makes establishing communication among them an involved task for programmers. The goal of Java's internetworking facilities is to hide the details of different physical networks from programmers. This allows the programmer to worry about the more romantic pursuits of network programming and not to be bogged down by the trivial details of many different systems. However, this grand achievement of hiding details was no walk in the park for the Java designers. Hosts can have vastly different physical attributes and may be dedicated to widely varying tasks. What is needed to make all these different species of systems happy and able to communicate with each other is a common protocol. A protocol is a set of rules and conventions between the communicating participants. Using the higher-level protocol abstractions, the programmer can create Java programs quickly and with increased productivity. They need not build special versions of application software to move and translate data between different types of machines.

This chapter introduces the basic concepts of networking. It discusses client-server applications, tells you how to identify a host using an Internet address, and explains what sockets are. Following a summary of the classes and interfaces covered in this chapter are detailed explanations of the methods presented with examples to help you build networking applications in Java. The project you will develop in this chapter is a client-server application. The client sends messages to the server requesting it to send the contents of a file. The server processes the request and sends the contents of the file line-by-line. On receiving the file's contents from the server, the client displays it on a window.

## Client-Server Applications

There are several models for building network applications. The most widely used model is the client-server model which involves two types of processes: a *server* process and a *client* process. When you start a server process on a host, it waits for a client to contact it. A client process, started on the same host or a different one, sends a request to the server over the network. The server responds to the request by sending a reply. Figure 10-1 illustrates a typical exchange in a client-server application.

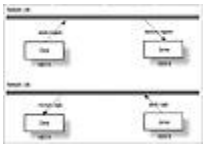

**Figure 10-1** A client-server communication scenario(a) Host A, as a client sends a request for service to server located on host B(b) After processing the request, the server sends a reply to the client

The communication between a server and a client can be accomplished in two ways: connection-oriented or connectionless. In a connection-oriented transfer, a dedicated connection is established between a server and a client. They use this connection to exchange information. Given that the other type is called connectionless, it doesn't seem like a lot of communication actually happens between them. Then how do they communicate? The client sends the request by specifying the server's address. This is received by the server, who is waiting for a message from some client. The server obtains the client address from the message to which it may then respond.

### Connection-Oriented Protocol

In a connection-oriented communication, the client and the server have a dedicated link established between them. It is similar to the telephone communication system. When you call someone and the called phone number exists, there is a dedicated line for you to converse. Whatever you speak is guaranteed to be heard on the other side, with probably an element of delay. Also the words you speak are heard in the exact order in which they were spoken. The connection-oriented protocol is a reliable protocol. The messages sent between any two processes are guaranteed to be delivered and in the proper sequence. Most of the networking applications are connection oriented, as they require reliable communication protocol. TCP (Transfer Control Protocol) is a connection-oriented protocol in the TCP/IP family.

### Connectionless Protocol

In a connectionless protocol, there is no dedicated link between the client and the server. They send messages as datagram packets, each of which contains the destination address. The underlying network will decipher the targeted destination address from a packet and routes the packet to the destination. In this sense, each packet is self-contained. They have the information about the sender and the intended receiver, apart from the core message. You can consider such a communication to be similar to the U.S. Postal service. Each letter you send has its destination address contained in it and the postal department takes the necessary steps to route the mail to the destination. But you should note that the postal department guarantees neither the delivery nor the sequence of delivery. Similarly, in the connectionless protocol, the packets are not guaranteed to be delivered, and even it they are delivered, the order of delivery is not guaranteed. Then the obvious question is: Where will you ever use the connectionless protocol for communication? You can use this protocol where the order of messages is not critical. For example, consider a time server application. The server can keep sending the updated time to the client. The client need not assume any order of delivery. As it receives messages, logic can be built in the

client application to sense the sequence of received messages. In this application, missing packets will not create any havoc. In the TCP/IP family, the UDP is the connectionless protocol.

## Internet Address

To identify a particular host in the Internet you need an Internet address. Using Internet addressing, a host can communicate with another host located in the same physical network or subnet, or in a different physical network, where both networks are linked by the Internet. Hence, hosts separated geographically can communicate effectively by addressing each other by their Internet address. Figure 10-2 illustrates the distribution of hosts among the Internet, divided into different physical networks.

**Figure 10-2** Distribution of hosts among different subnets over the Internet

An Internet address is usually written as four decimal numbers, which are separated by decimal points. Each decimal digit in this address encodes one byte of the 32-bit Internet address. The Internet address maps to a unique host and the host can be addressed by a unique combination of the hostname and the name of the particular network the host is a part of (i.e., domain name). Conceptual representation of Internet addresses is represented in Figure 10-3.

**Figure 10-3** Conceptual representation of Internet addressing

In the Internet address representation, Internet Id identifies a network in the Internet. Subnet Id represents a local area network(*subnet*) and Host Id is the identifier for a given host in the subnet. The combination of these three identifiers represents a unique host in the Internet. Here 128.230.32.66 is the Internet address that maps to ratnam.cat.syr.edu, where ratnam is the machine name and is a part of cat.syr.edu network domain. In this example, host Id is 66, subnet Id is 32 and the Internet Id is 128.230. The InetAddress Class in Java encapsulates the methods required to manipulate with an Internet address in a networking application.

## Why Sockets?

To provide an interface between your application and the network you need a socket. Most of the communication in a client-server application is point-to-point, where the endpoint of such communication is an application (client or server). A socket acts as an

endpoint for communication between processes on a single system or on different systems. The applications communicate between themselves by sending messages to one another. These messages are sent as a sequence of packets at the network level. For each packet that is sent, there has to be a receiving end. Sockets form such an end point to receive packets, as well as to send messages. Application programs request the operating system to create a socket when in need. The system returns a socket identifier, in the form of a small integer that the application program uses to reference the newly created socket. A networking application can be identified by a <host, socket> pair (the host on which it is running and the socket at which it is listening for messages).

Java provides separate classes which encapsulate the functionality of client and server sockets. The Socket class is used to represent sockets on the client side, while the server side sockets are represented by the ServerSocket class. The Socket and ServerSocket form the client and server side sockets in a connection oriented protocol. Once a link is established between the client and the server, they can exchange messages until one of them closes the connection. Whereas, the sockets, in the case of connectionless protocol, are represented by the DatagramSocket class. In this case, both the client and the server are associated to a datagram socket. Every time a message is to be sent, they create a DatagramPacket containing the destination address and port number along with the message to be sent. This packet gets delivered (if it does get delivered) to the targeted application. To implement various such policies for communication between a client and a server, Java provides a SocketImpl Class. SocketImplFactory is an Interface that can be used to generate more instances of the SocketImpl Class for use in your applications.

## Network and Socket Summaries

Table 10-1 summarizes the classes and interfaces necessary for developing network applications using Java.

**Table 10-1** Class and interface description

| Class/Interface | Description |
|---|---|
| InetAddress | Represents Internet addresses. |
| ServerSocket | This class represents the server socket. It uses a SocketImpl class to implement the actual policies regarding socket operations. |
| Socket | This class represents the client socket. It uses a SocketImpl class to implement the actual socket policies for its operations. |
| DatagramSocket | This class represents a datagram socket, which is an implementation of the connectionless protocol. |
| DatagramPacket | This class represents a datagram packet, which is self-contained, |

| | |
|---|---|
| | with the details of the destination host and the data to be sent. |
| SocketImpl | This should be subclassed to provide actual implementation. This class implements the actual socket policies for ServerSocket and Socket classes. |
| SocketImplFactory | A factory for creating actual instances of SocketImpl is defined in this interface. |

## InetAddress

**Purpose**

Use InetAddress to represent Internet addresses.

**Syntax**

public final class InetAddress extends Object

**Description**

The InetAddress class represents the Internet Address. The methods of InetAddress provide functionality to gain information about raw IP address, hostname, and network address of a host machine, and hash code of the Internet address in the hashtable. Figure 10-4 shows the inheritance diagram for the InetAddress class.

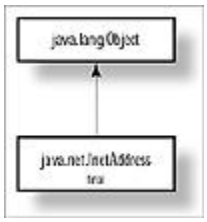


**Figure 10-4** Class diagram for InetAddress class

**PackageName**

*java.net*

**Imports**

*import java.net.InetAddress;*

**Constructors**

None; see getByName(*String*) method

**Parameters**

None

**Example**

Use the getByName(*String*) method of the InetAddress class to create an instance of InetAddress. No public constructors are provided for InetAddress, so an instance of InetAddress cannot be created using the *new* statement. In the following example, the Client obtains the InetAddress of the server to connect to. It uses the hostname of the server to create an InetAddress object.

```
import java.net.InetAddress;

public class Client extends Applet implements Runnable {
```

```
        Socket sock;            // The Client Socket object

                // Client class members
                String servHost;        //string containing the server
                                        hostname
                int servPort;   // port number on which the server is
                                listening to

                                // other members

        public void init() {

        /* Obtain the server hostname using command line arguments or
        from web page */
        // code omitted
        // :
        // :
       // obtain Inetaddress of server by specifying the server's
       hostname
       servHost = new String("serval.cat.syr.edu");
                InetAddress addr = InetAddress.getByName(servHost);
       sock = new Socket(addr, servPort); //create a client socket

       /* Other actions */

       #125; // end of init()
}
```

## equals(*Object*)

**ClassName**

InetAddress

**Purpose**

Compares the specified object with the object on which the method is invoked.

**Syntax**

public boolean equals(Object objct)

**Parameters**

***objct***

The object with which the invoked object is to be compared.

**Description**

The method compares the Internet address of the specified object with that of the object on which the method is invoked. The objects are considered equal if their Internet addresses are the same.

**Imports**

*import java.net.InetAddress;*

**Returns**

The return type of the method is boolean. It returns true if the objects compared are the same; if the objects are not the same, the method returns false.

**See Also**

class InetAddress

**Example**

The following example obtains the InetAddress of both the Client and the Server. It compares these objects to verify whether the Client is running on the same host as the Server by using the equals(object) method of InetAddress class. The result is displayed.

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

public class Client extends Applet implements Runnable {

 // Client class members
       Socket sock;

 /* other members and methods
         public void printOut(String str){ ..}
         public void init(){ ..}
         :
  */

   public void compareHosts() {
       InetAddress c_inet =       InetAddress.getLocalHost();
       InetAddress s_inet =       sock.getInetAddress();

      boolean sameHost = c_inet.equals(s_inet);
      if (sameHost) /* returns TRUE */
          printOut(" Client and Server run on the same host");
      else  /* returns FALSE*/
          printOut(" Client and Server run on different Hosts");
   }
  }
```

## getAddress()

**ClassName**

InetAddress

**Purpose**

This method returns the raw IP address of the object in network byte order.

**Syntax**

public byte[] getAddress()

**Parameters**

None.

**Description**

The method returns the raw IP address representation of the Internet address in 32-bit format in network byte order. It returns the addr[] byte array, member of the InetAddress. addr[0] contains the highest order byte position. addr[] is an array of bytes so that this method is extendable for 64-bit IP addresses also.

**Imports**

*import java.net.InetAddress;*

**Returns**

The value of the addr[] byte array in InetAddress; the return type is a byte array.

**See Also**

The class InetAddress

**Example**

The following example gets the raw IP address of machine on which the Server is running, servHost, into a byte array.

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

public class Client extends Applet implements Runnable {
/*
       Members and methods of the class
       :
       :

 */

 public  void RawIp() {
         InetAddress inet = InetAddress.getByName(servHost);

         /* gets the raw IP address in network byte order */
         byte[] raw_ip = inet.getAddress();
  }
}
```

## getAllByName(*String*)

**ClassName**

InetAddress

**Purpose**

Returns an array of all InetAddresses that correspond to the specified hostname.

**Syntax**

public static synchronized InetAddress[] getAllByName(String host_name) throws UnknownHostException

**Parameters**

***host_name***

The hostname of the machine, the InetAddresses which you are trying to obtain.

**Description**

A host can have multiple InetAddresses (Internet address mapping). To access all of those InetAddresses, the hostname is passed to the getAllByName method. The method finds out the Internet addresses of the given host and returns all of them as an array of InetAddress objects.

**Imports**

*import java.net.InetAddress;*

**Returns**

The return type of this method is an array of InetAddresses. The array elements contain all the InetAddresses of the specified host.

**See Also**

The class InetAddress; method InetAddress.getByName(String hostname)

**Example**

In the following example, the Client obtains all the InetAddresses of the Server host into an array of InetAddress.

```
import java.net.InetAddress;

public class Client extends Applet implements Runnable{
       // Client class members and methods
       // :
       // :

public void ServerInfo(){
       // String servHost; contains host name of Server
       // more code here …

InetAddress[] inet =
        InetAddress.getAllByName(servHost);
}
}
```

## getByName(*String*)

**ClassName**

InetAddress

**Purpose**

This method gets the InetAddress of the specified host.

**Syntax**

public static synchronized InetAddress getByName(String host_name) throws UnknownHostException

**Parameters**

***host_name***

The hostname of the machine whose InetAddress is returned by this method.

**Description**

This method returns the InetAddress of a specified host. The InetAddress class does not have public constructors. You can use this method to create an instance of the InetAddress for a particular host.

**Imports**

*import java.net.InetAddress;*

**Returns**

The return value of the method is InetAddress. It returns the InetAddress for the specified host.

**See Also**

The class InetAddress; method InetAddress.getAllByName(String)

**Example**

Refer to the code example in the class InetAddress API where the method getByName is used to obtain the InetAddress of the server host by specifying the server's hostname.

## getHostName()

**ClassName**

InetAddress

**Purpose**

This method returns the hostname for this InetAdress.

**Syntax**

public String getHostName()

**Parameters**

None

**Description**

The method returns the hostname of a machine with the same address as this InetAddress. So if you know the IP address of a machine, you can find out its hostname by using this method on the InetAddress object of that address.

**Imports**

*import java.net.InetAddress;*

**Returns**

The return type of the method is String and its value is the hostname of the machine with this IP address.

**See Also**

The class InetAddress

**Example**

The following example illustrates the usage of the getHostName() method to obtain the hostname of the Client.

```
import java.net.InetAddress;

public class Client extends Applet implements Runnable{

        // class members and other methods
        // code omitted
  public void ClientInfo(){
        InetAddress c_inet;
        String c_name;
        try {
          c_inet = InetAddress.getLocalHost();
          c_name = InetAddress.getHostName();
                // obtain the string form of InetAddress
          String c_str = c_inet.toString(); //obtained by using
          getLocalHost()

         // get the index where '/' appears; before that
         // character is the IP address in the InetAddress String
           int  index = c_str.indexOf('/');

           String c_ipaddr = c_str.substring(index+1); //obtain the ip
           address only
           printOut(" IP Address : " + c_ipaddr);

           // :
           // :
         } catch (IOException ioE);

  }
}
```

## getLocalHost()

**ClassName**

InetAddress

**Purpose**

This method gets the InetAddress of the local host.

**Syntax**

public static InetAddress getLocalHost() throws UnknownHostException

**Parameters**

None

**Description**

This method finds the Internet address of the local machine executing this program. It creates an instance of InetAddress with this address and returns the InetAddress object. getLocalHost() can be used to create an instance of InetAddress for the local machine.

**Imports**

*import java.net.InetAddress;*

**Returns**

The return type of the method is InetAddress. It returns the InetAddress object representing the Internet address of the local machine.

**See Also**

The class InetAddress; method InetAddress.getByName()

**Example**

Refer to the example in getHostName() which contains a call to this method.

## hashCode()

**ClassName**

InetAddress

**Purpose**

Returns the hash code of this InetAddress object.

**Syntax**

public int hashCode()

**Parameters**

None

**Description**

The method returns the hash code, to be used as an index into the hashtable to access this InetAddress object. All the InetAddresses accessed during the program execution are cached in a hashtable. This is done for faster access of previously accessed Internet addresses.

**Imports**

*import java.net.InetAddress;*

**Returns**

The return type of this method is int, and the value is the hash code of this Internet address.

**See Also**

The class InetAddress

**Example**

The following example gets the hash code for the InetAddress object of the server host.

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

public class Client extends Applet implements Runnable {
        // class members and methods

  public void ServerInfo() {
        // other code
        InetAddress inet =
                InetAddress.getByName(servHost);
    /* get and prints the hash code of the above inet object */
        int hash = inet.hashCode();
        printOut ("Hash Code for server is: " + hash);
  }
}
```

## toString()

**ClassName**

InetAddress

**Purpose**

This method converts the InetAddress to a string.

**Syntax**

public String toString()

**Parameters**

None

**Description**

This method converts the InetAddress to a String by overriding the toString() method of the Object class. Raw IP address, host name can also be obtained by manipulating with the returned String.

**Imports**

*import java.net.InetAddress;*

**Returns**

The return type of the method is String. It returns the String form of the InetAddress.

**See Also**

The class InetAddress, method InetAddress.getHostName()

**Example**

Refer to the code example for getHostName() method of InetAddress class.

## ServerSocket

**Purpose**

Use ServerSocket to implement a server.

**Syntax**

public final class ServerSocket extends Object

**Description**

The ServerSocket class represents the server in a client-server application. This class implements the actual socket policies that go along with a server. It uses a default SocketImpl class to implement its server policies. These policies can be changed by implementing a concrete subclass of the abstract SocketImpl class. This change in policies can be made effective by setting the SocketImplFactory, using the setSocketFactory method. The methods of ServerSocket class provide the functionality to create a server socket, accept connection from a client and get the specifics of the particular ServerSocket object (namely the port to which the server is connected), and the string form of implementation address, file descriptor, and port. A ServerSocket object is bound to the local machine on which it is created. A port number is specified for the ServerSocket to bind and listen for connections. Figure 10-5 shows the inheritance diagram for the class ServerSocket.

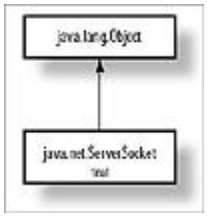


**Figure 10-5** Class diagram for ServerSocket class

**PackageName**

*java.net*

**Imports**

*import java.net.ServerSocket;*

**Constructors**

public ServerSocket(int *port*) throws IOException

public ServerSocket(int *port*, int *lisn_time*) throws IOException

**Parameters**

***port***

Local port number to which the server binds.

***lisn_time***

The amount of time the server will listen at the port for connection.

**Example**

In the following example, an instance of ServerSocket is created and is bound to the port number specified by *port*. The listening time for a connection is set to 10. After creation, the socket waits for a connection from a client and accepts the connection. After performing its services, the server socket is closed.

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.*;
```

```
public class Server {

public static void main(String args[]) {
        int port; // port number
             try {
        // obtain the port number or set a default
        // :
        // create  a server socket object

        ServerSocket serv = new ServerSocket(port, 10);
  } catch (IOException io) {
        System.out.println(" Error: Creating a server socket");

  }
  try {
       /* Accept a connection from a client */
       Socket clnt = serv.accept();
  } catch (IOException io) {
       System.out.println(" Error: Accepting a connection ");
  }

       // get the string form of the server socket
  String str = serv.toString();
  System.out.println("String form of server socket is : " + str);

  /*  perform the services */

   /* Close the socket connection now */
  try {
      serv.close();
  } catch (IOException io) {
      System.out.println(" Error: closing the server socket ");
  }
       } /* end of main() */
} /* end of class Serv */
```

## accept()

**ClassName**

ServerSocket

**Purpose**

Blocks until a connection is made with a client. It returns a client socket after the connection is established. This socket is used for further communication with the client.

**Syntax**

public Socket accept() throws IOException

**Parameters**

None.

**Description**

This method blocks for a client connection by listening to the port it has bound to. When a client attempts to connect to this server socket, this method accepts the connection and returns a client socket (instance of the Socket class). Optionally, the time limit for listening can also be specified. If it is specified during construction, then the server blocks for connection for only the specified time. This socket is later used for all the communication between the server and its client. The output stream of this socket is the input for the connected client and vice versa. An IOException is thrown if any error has occurred while establishing the connection. This has to be caught and relevant exception handling steps should be taken.

**Imports**

*import java.net.ServerSocket;*
*import java.net.Socket;*

**Returns**

The return type of the method is a Socket. It returns the Socket instance created on the server's side to communicate with the connected client.

**See Also**

The class ServerSocket; class Socket

**Example**

Refer to the example for class ServerSocket, given above. In that example, an instance server of ServerSocket is created and, after creation, the socket waits for a connection from a client and accepts the connection using the accept() method.

## close()

**ClassName**

ServerSocket

**Purpose**

Closes the socket of the server.

**Syntax**

public void close() throws IOException

**Parameters**

None.

**Description**

The method closes the socket to which the server is bound. This implies that any other client socket connected to this server should have been closed prior to this method call, for proper behavior. So before calling the close method on a server, all the clients connected to it should have closed their socket connections.

**Imports**

*import java.net.ServerSocket;*

**Returns**

None.

**See Also**

The class ServerSocket; Exception IOException

**Example**

Refer the example for the API class ServerSocket

## getInetAddress()

**ClassName**

ServerSocket

**Purpose**

Returns the InetAddress object, representing the Internet address of the host to which the server socket is bound.

**Syntax**

public InetAddress getInetAddress()

**Parameters**

None.

**Description**

This method returns the InetAddress of the host machine on which the ServerSocket is bound. As a server is bound only to a local machine (optionally specifying a port number), this method helps you identify the host to which the server is created and uses this InetAddress object to access more information about the server host.

**Imports**

*import java.net.ServerSocket;*
*import java.net.InetAddress;*

**Returns**

The return type of the method is InetAddress. It returns the InetAddress object which represents the Internet Address of the host machine on which the server socket is created and bound.

**See Also**

The class ServerSocket; class InetAddress

**Example**

In the following example, an instance of ServerSocket is created. InetAddress of the server host is obtained and hostname, IP address details of that host are retrieved and printed.

```
import java.net.ServerSocket;
import java.net.InetAddress;
import java.io.*;

public class Server {

public static void main(String args[]) {
             try {
         ServerSocket serv = new ServerSocket(5000, 10);
   } catch (IOException io) {
         System.out.println("Error: Creating a server socket");
   }
         /* get the InetAddress of the server host */
   InetAddress inet = serv.getInetAddress();

         /* Get details of the server host */

   String addr_str = inet.toString();
   byte[] raw_ip =   inet.getAddress();
   System.out.println("The InetAddress of server host is : "
```

```
   + addr_str);

        /* get the port number of the server socket */
   int port_num = serv.getLocalPort();

   if (port_num == 3001)
        System.out.println("Server port# is 3001 as expected ");
   else
        System.out.println("Server port# is not 3001!  ;-( ");

        /* close the socket */
   try {
       serv.close();
   } catch (IOException io) {
   }
       } /* end of main() */
} /* end of class Serv */
```

## getLocalPort()

**ClassName**

ServerSocket

**Purpose**

Returns the port number to which the server socket connection is established and is listening.

**Syntax**

public int getLocalPort()

**Parameters**

None.

**Description**

This method obtains the port number to which the ServerSocket is bound and listens for connection. This number is the port number of the socket at the machine on which the ServerSocket is connected.

**Imports**

*import java.net.ServerSocket;*

**Returns**

The return type of the method is int. It returns the port number of the ServerSocket instance created at the server's side.

**See Also**

The class ServerSocket

**Example**

Refer to the example in the getInetAddress() method of ServerSocket class.

## setSocketFactory(*SocketImplFactory*)

**ClassName**

ServerSocket

**Purpose**

Sets the system's server SocketImplFactory interface, which generates SockImpl instances.

**Syntax**

public static synchronized void
setSocketFactory(SocketImplFactory fac) throws IOException

**Parameters**

None.

**Description**

The desired policies that go with the ServerSocket class are implemented by a concrete subclass of SocketImpl. Instances of this subclass will be created by the createSocketImpl() method of the SocketImplFactory interface. When *fac*, an instance of this interface, is passed to this setSocketFactory() method, the member factory of ServerSocket class gets assigned. Thereafter, any instance of ServerSocket created will have the desired socket policies implemented. The SocketImplFactory can be specified only once for this ServerSocket class. This method will not normally be used by programmers, as the APIs for ServerSocket have a default SocketImpl defined and, hence, there is no need for a factory.

**Imports**

*import java.net.ServerSocket;*

**Returns**

None.

**See Also**

The class ServerSocket; class SocketImpl; interface SocketImplFactory

**Example**

The following code outline illustrates how to subclass SocketImpl and use it to generate instances of that class using a factory. This factory, MyFac, is set to be the member factory of server class Server so that any new instances of Server will implement the desired policies as in MySockImpl.

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

class MySockImpl extends SocketImpl {

/* Implement all the methods of SocketImpl class to adhere by the
desired server socket policies */

}

/** MyFac is the concrete class implementing SocketImplFactory for
this application */

class MyFac implements SocketImplFactory {

         SocketImpl createSocketImpl() {
                   /* return an instance of MySockImpl */
                   return new MySockImpl();
         }
}
```

```
/** Assign MyFac as the factory member of class Serv, representing a
server socket */

public class Server {

  public static void main(String args[]) {
          SocketImplFactory fac = new MyFac();
          ServerSocket.setSocketFactory(fac);

/* Any new instances of ServerSocket created hereafter will implement
   the policies defined by */
   MySockImpl class */

          } /* end of main() */
} /* end of class Serv */
```

## toString()

**ClassName**

ServerSocket

**Purpose**

To obtain the implementation address, file descriptor, and port number to which this ServerSocket is connected.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method creates a String object containing the port number, file descriptor, and address to which the ServerSocket is bound to. The details are merged into string form within a String object. This method overrides the toString() method in the class Object.

**Imports**

*import java.net.ServerSocket;*

**Returns**

The return type of the method is String. The returned String object contains the port number, file descriptor, and address of the machine on which the SeverSocket is bound.

**See Also**

The class ServerSocket; class Object (java.lang.Object)

**Example**

Refer to the example in ServerSocket class description.

## Socket

**Purpose**

Use Socket to implement a client to communicate with a server.

**Syntax**

public final class Socket extends Object

**Description**

The Socket class represents a client in a client-server application. This class implements the actual socket policies that go along with a client. It uses a default SocketImpl class to implement its client socket policies. These policies can be changed by implementing a concrete subclass of the abstract SocketImpl class. This change in policies can be made effective by setting the SocketImplFactory, using the setSocketImplFactory method. The methods of the Socket class provide the functionality to create a client socket, by connecting to a server, and to obtain necessary input and output streams for communication. The class contains methods to find out the specifics of the particular Socket object namely, the port to which the client is connected, the string form of implementation address, and the port number. Figure 10-6 shows the inheritance diagram of the Socket class.

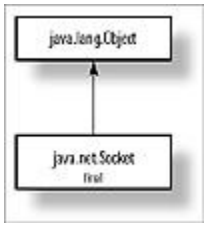


**Figure 10-6** Class diagram for Socket class

**PackageName**

*java.net*

**Imports**

*import java.net.Socket;*

**Constructors**

public Socket(String *host*, int *port*) throws IOException
public Socket(String *host*, int *port*, boolean *sock_type*) throws IOException
public Socket(InetAddress *inet*, int *port*) throws IOException
public Socket(InetAddress *inet*, int *port*, boolean *sock_type*) throws IOException

**Parameters**

***host***

Name of the host machine to connect to.

***port***

Local port number to which the server binds.

***sock_type***

Value specifying whether it is a datagram socket or a stream socket.

***inet***

InetAddress of the host machine to connect to.

**Example**

In the following example, an instance of Socket is created in the class Client. A server is initially created in the Server class. The client is connected to the server in its init() method. If the server does not exist when the client attempts a connection, an exception is thrown to indicate unavailable server connection. So the server should be started before any client attempts to request service.

```
import java.net.ServerSocket;
import java.net.Socket;
```

```
import java.io.*;

/* Server class */

public class Server {

public static void main(String args[]) {2
             try {
                // create a server socket

         ServerSocket serv = new ServerSocket(3001, 10);
   } catch (IOException io) {
         System.out.println(" Error: Creating a server socket");
   }
   try {

         Socket clnt = serv.accept(); // accept connections from clients
   } catch (IOException io) {
         System.out.println(" Error: Accepting connection ");
   }
   /*  perform the services */

   try {
         serv.close(); // close the socket connection
   } catch (IOException io) {
         System.out.println(" Error: Closing the server socket ");
   }
}
}

/* Client class implementation */

public class Client extends Applet implements Runnable {
        // members and methods of class Client
/*

 */

   public void init(){
       int servPort = 3001;
       String servHost = "serval.cat.syr.edu";
         //obtain the server host name in this String object

   try {

         Socket clnt = new Socket(servHost, servPort);
         String sock_str = clnt.toString();
                              // usage of toString() method
         System.out.println(" String form of Socket is " + sock_str);
         // print the
         // string
         // containing
         // the socket
         // details

     } catch (IOException io) {
         System.out.println("Error: Connecting to the server");
```

```
    }
   } // end of init()
            /*  request services from the server */

   public void run() {
        // interact with the server

   }

        // Applet stop
public void stop() {
     try {  // use of close() method of Socket class
              sock.close();
         } catch (IOException ioE) {
               System.out.println(" Error in socket.close()");
              }
   }
} // end of class Client
```

## close()

**ClassName**
Socket
**Purpose**
Closes the socket of the client.
**Syntax**
public void close() throws IOException
**Parameters**
None.
**Description**
The method closes the client's socket connection to the server.
**Imports**
*import java.net.Socket;*
**Returns**
It does not return anything. It throws an IOException on an occurrence of an error, which has to be caught.
**See Also**
The class Socket; Exception IOException
**Example**
Refer to the example in Socket class which includes the usage of close() method.

## getInetAddress()

**ClassName**
Socket
**Purpose**
Returns the InetAddress object, representing the Internet address of the host to which the client socket is connected, i.e., the server host.
**Syntax**

public InetAddress getInetAddress()

**Parameters**

None.

**Description**

This method returns the InetAddress of the host machine on which the Socket is connected. As a client connects to a server that is already ready and running, this method helps you identify the host on which the server is created and use this InetAddress object to access more information about the server host.

**Imports**

*java.net.Socket;*
*import java.net.InetAddress;*

**Returns**

The return type of the method is InetAddress. It returns the InetAddress object which represents the Internet Address of the host machine to which the client has established connection.

**See Also**

The class Socket; class InetAddress

**Example**

Refer to the ServerInfo() method of Client class implemented in the Client-Server Rendezvous applet at the end of this chapter.

## getInputStream()

**ClassName**

Socket

**Purpose**

Returns an InputStream for this client socket.

**Syntax**

public InputStream getInputStream() throws IOException

**Parameters**

None.

**Description**

This method returns an InputStream for this client socket. The client receives data from the server using this stream. It can block on this InputStream until data arrives from the server. This method is also used on the server side, when the server obtains a Socket object on accepting a connection from a client. An InputStream of client is bound to the OutputStream for the Socket on server side and vice versa.

**Imports**

*import java.net.Socket;*

**Returns**

The return type of the method is InputStream. It returns an InputStream for the client. This object can further be used to instantiate another kind of stream suitable for service provided by the server.

**See Also**

The class Socket; class InputStream

**Example**

Usage of getInputStream() is important for client-server applications as this is the method such applications use for establishing stream channels to pass messages. The server creates an InputStream to obtain data from the client by using the Socket object, returned by the accept() call. The client obtains an InputStream from its Socket, which is effectively bound to the server's OutputStream; likewise, the client's OutputStream is bound to the InputStream of the server. Refer to the init() method of Client class in the Client-Server Rendezvous applet at the end of this chapter.

## getOutputStream()

**ClassName**

Socket

**Purpose**

Returns an OutputStream for this client socket.

**Syntax**

public OutputStream getOutputStream() throws IOException

**Parameters**

None.

**Description**

This method returns an OutputStream for this client socket. The client sends data to the server using this stream. This method is also used on the server side, when the server sends a message to the client. An OutputStream of client is bound to InputStream for the Socket on server side and vice versa.

**Imports**

*import java.net.Socket;*

**Returns**

The return type of the method is OutputStream. It returns an OutputStream for the client. This object can further be used to instantiate another kind of stream, suitable for service provided by the server.

**See Also**

The class Socket; class OutputStream

**Example**

Refer to the init() method of Client class in the Client-Server Rendezvous applet at the end of this chapter.

## getLocalPort()

**ClassName**

Socket

**Purpose**

Returns the local port number to which the client socket is connected.

**Syntax**

public int getLocalPort()

**Parameters**

None.

**Description**

This method obtains the local port number to which the Socket is connected. This number is the port number to which the client socket is connected on the local(client) side.

**Imports**

*import java.net.Socket;*

**Returns**

The return type of the method is int. It returns the port number to which the Socket has connected.

**See Also**

The class Socket

**Example**

In the following example, an instance of ServerSocket is created and is bound to the port number 3001. In Class Client, a client socket is created using the Socket instance and connection is established to the server. The getLocalPort() method is used to obtain the local port number.

```
/* Client class implementation */

public class Client {

 public static void main(String args[]) {
   /* String servHost contains server hostname */
             try {
       Socket clnt = new Socket(servHost, 3001);
   } catch (IOException io) {
        System.out.println(" Error: Creating a client socket");
   }

  /* Obtain the port number to which the socket is connected */
   int port_num = clnt.getLocalPort();
   printOut(" Local port number of client is : " + port_num);

   /* Close the connection from client side */

   try {
        clnt.close();
   } catch (IOException io) {
        System.out.println(" Error: Closing the connection");
   }
         } /* end of main() */
} /* end of Client class definition */
```

## getPort()

**ClassName**

Socket

**Purpose**

Returns the port number on the server side to which the client has established connection.

**Syntax**

public int getPort()

**Parameters**

None.

**Description**

This method obtains the port number to which the Socket is connected. This number is the port number on which the server is listening for connections and to which this client Socket object has established connection.

**Imports**

*import java.net.Socket;*

**Returns**

The return type of the method is int. It returns the port number to which the Socket has established connection with the server.

**See Also**

The class Socket

**Example**

The ClientInfo() method of Client class in the Client-Server Rendezvous applet at this chapter's end illustrates the use of getPort() method.

## setSocketImplFactory(*SocketImplFactory*)

**ClassName**

Socket

**Purpose**

Sets the system's client SocketImplFactory interface, which generates SockImpl instances.

**Syntax**

public static synchronized void setSocketImplFactory(SocketImplFactory fac) throws IOException

**Parameters**

None.

**Description**

The desired policies that go with the Socket class are implemented by a concrete subclass of SocketImpl. Instances of this subclass will be created by the createSocketImpl() method of the SocketImplFactory interface. When *fac*, which is an instance of this interface, is passed to this setSocketImplFactory() method, the member factory of ServerSocket class gets assigned. Thereafter, any instance of Socket created will have the desired socket policies implemented. This method will not be normally used by programmers, as the APIs for Socket have a default SocketImpl defined and hence, there is no need for a factory.

**Imports**

*import java.net.Socket;*

**Returns**

None.

**See Also**

The class Socket; class SocketImpl; interface SocketImplFactory

**Example**

In the following example, class MySockImpl is implemented by subclassing SocketImpl. Instances of this class are generated by using MyFac, an implementation of SocketImplFactory. The Client's factory member is set to an instance of this factory and, thereafter, all instances of Socket are defined by the implementation in MySockImpl.

```
import java.net.ServerSocket;

class MySockImpl extends SocketImpl {

/* Implement all the methods of SocketImpl class to adhere by the
desired client socket policies */

}

class MyFac implements SocketImplFactory {

         SocketImpl createSocketImpl() {
                  /* return an instance of MySockImpl */
         }
}

public class Client  extends Applet implements Runnable {

  public static void main(String args[]) {
         SocketImplFactory fac = new MyFac();
         Socket.setSocketImplFactory(MyFac);

/* Any new instance of Socket created hereafter   will implement the
policies defined by MySockImplclass */

         } /* end of main() */
} /* end of class Client */
```

## toString()

**ClassName**

Socket

**Purpose**

To obtain the implementation address, file descriptor, and port number to which this Socket is connected.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method obtains a String object containing the port number, local port number, and address to which the Socket is bound. The details are merged into string form within a String object. This method overrides the toString() method in the class Object.

**Imports**

*import java.net.Socket;*

**Returns**

The return type of the method is String. The returned String object contains the port number to which the socket is connected, local port number of the socket, and address of the machine on which the Socket is connected.

**See Also**

The class Socket; class Object (java.lang.Object)

**Example**

Please refer to the example in the Socket class description which shows the details of the Socket in string form.

## DatagramSocket

**Purpose**

Use the DatagramSocket class when you develop applications based on connectionless protocol.

**Syntax**

public class DatagramSocket extends Object

**Description**

An instance of this class forms a socket in a client-server application which is implemented using the connectionless protocol. It uses the UDP/IP protocol over the Internet. This is the class that should be instantiated on both the server and the client side, unlike ServerSocket and Socket implementations in connection-oriented protocol. When you create a DatagramSocket object on the server side, you can use the contructor that takes the port number as an argument. This way your server listens for messages at the specified port. The client's port number need not be specified. The client sends a packet to the server by specifying the port number when creating the datagram packet. This class contains the send and receive methods for handling the packets. Other methods are provided for closing a connection, getting the local port number and for cleaning up purposes. Figure 10-7 illustrates the inheritance diagram for the DatagramSocket class.

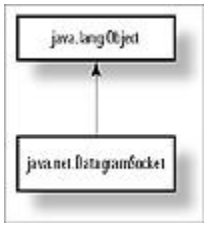


**Figure 10-7** Class diagram for DatagramSocket class

**PackageName**

*java.net*

**Import**

*import java.net.DatagramSocket;*

**Constructors**

public DatagramSocket() throws SocketException

public DatagramSocket(int *port*) throws SocketException

**Parameters**

*port*

The local port number at which the datagram socket is bound.

**Example**

Refer to Listing 10-1 in which a message server and a message client are defined. In the constructor of the messageServer class, a datagram socket is constructed by specifying the port number 3000. That is the port at which the server waits for messages. Any client application should send a message to that port number, which is 3000 in this case, if it wants to communicate with the server. Note that in the constructor of the messageClient class (Listing 10-2), the port is not specified.

**Listing 10-1** messageServer.java: A datagram server class that sends messages to clients

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

public class messageServer {

        private DatagramSocket serverSocket;
        private int messageNumber;

        public messageServer() {
                try {

                        serverSocket = new DatagramSocket(3000);
                        System.out.println("Local port is " +
                        serverSocket.getLocalPort());
                        messageNumber = 0;
                } catch (SocketException se) {
                        System.out.println(" Error creating datagram
                        socket");
                }
}
        public void run() {
                while (true)
                        processRequests();
        }

        public void processRequests() {
                try {
                  byte[] buffer = new byte[256];
                  DatagramPacket packet = new DatagramPacket(buffer,
256);
                  serverSocket.receive(packet);
```

```
                  InetAddress clientAddr = packet.getAddress();
                  int clientPort = packet.getPort();
                  String msg = new String(" This is message #" +
                  messageNumber++);
                  // convert the string to an array of bytes
                  msg.getBytes(0, msg.length(), buffer, 0);

                  // construct a new datagram packet to send to the
client

                  packet = null;
                  packet = new DatagramPacket(buffer, buffer.length,
                  clientAddr, clientPort);
                  // send the packet to the client
                  serverSocket.send(packet);

                } catch (Exception e) {
                        System.out.println("Exception in
                        processRequests");
                }
        }

        public void finalize() {
                if (serverSocket != null) {
                        serverSocket.close();
                        serverSocket = null;
                }
        }

        public static void main(String args[]){

                messageServer ms = new messageServer();
                ms.run();
        }
}
```

**Listing 10-2** messageClient.java: A datagram client class that receives messages from servers

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

public class messageClient {

        public static void main(String args[]){
          int serverPort;
          InetAddress serverAddr;

          DatagramSocket clientSocket;
          DatagramPacket packet;
          byte[] buffer = new byte[256];

          try {

                //create the client socket
```

```
                clientSocket = new DatagramSocket();

                serverPort = 3000;
                serverAddr =
InetAddress.getByName("serval.cat.syr.edu");

                int i=0;
                while (i++ < 10 ) {
                packet = new DatagramPacket(buffer, 256, serverAddr,
                serverPort);
                clientSocket.send(packet);

                clientSocket.receive(packet);
                String msg = new String(packet.getData(), 0);


                if (i==1)
                System.out.println(" Packet is recd from: " +
                                packet.getAddress().getHostName()
                                + " at port " + packet.getPort()
                                + " and the length is " +
                                packet.getLength());
                System.out.println(" Client received => " + msg);
                packet = null;
                }
                clientSocket.close();
                clientSocket = null;
                serverAddr = null;
          } catch (Exception e) {
                System.out.println(" Exception occurred on client
side");
          }
       }
}
```

## close()

**ClassName**

DatagramSocket

**Purpose**

Closes the datagram socket connection.

**Syntax**

public synchronized void close()

**Parameters**

None

**Description**

This method is used when a datagram socket connection is to be closed. Usually this is done during the clean-up after a client-server communication session ends. The clients should close the connection before the server.

**Imports**

*import java.net.DatagramSocket;*

**Returns**

None.

**Example**

Refer to Listing 10-1 where the close method is invoked on the server inside the finalize method. Before the garbage collector cleans up the object, the socket connection is closed. In Listing 10-2, the client closes the socket connection after receiving ten messages from the server.

## finalize()

**ClassName**

DatagramSocket

**Purpose**

This method gets invoked before the DatagramSocket object is garbage collected.

**Syntax**

protected synchronized void finalize()

**Parameters**

None

**Description**

This method is invoked by the garbage collector Thread before cleaning up the object. You can write your clean-up routines, such as closing socket connection, closing streams, and so on, in this method. This method overrides the finalize method of the class Object.

**Imports**

*import java.net.DatagramSocket;*

**Returns**

None.

**Example**

In Listing 10-1, the server socket is closed and assigned a null value in the finalize method.

## getLocalPort()

**ClassName**

DatagramSocket

**Purpose**

Obtains the port number of the datagram socket.

**Syntax**

public int getLocalPort()

**Parameters**

None

**Description**

This method returns the port number at which the socket is bound. This number will form a part of any packet that is sent to another datagram socket. If a message is to be sent from a client, the local port number of the server socket has to be specified when creating the packet to be sent.

**Imports**

*import java.net.DatagramSocket;*

**Returns**

The port number; return type is int.

**Example**

In Listing 10-1, the local port of the server socket is obtained by using this method. The port number 3000, the port at which the server is waiting, is displayed on the screen.

## receive(DatagramPacket)

**ClassName**

DatagramSocket

**Purpose**

This method receives the datagram packet that is sent to the target DatagramSocket object.

**Syntax**

public synchronized void receive(DatagramPacket packet)

**Parameters**

***packet***

The packet that is to be received by the target DatagramSocket object.

**Description**

This method receives the packet sent to the DatagramSocket object. When invoked, this method blocks until some input is available. The packet contains the buffer of bytes (data), packet length, the destination InetAddress, and port number. Using this method along with the send method, packets are exchanged between a client and a server. Only one Thread can be invoking a receive on a socket object as the receive method is synchronized.

**Imports**

*import java.net.DatagramSocket;*

**Returns**

None.

**Example**

In Listing 10-1, the server socket receives a datagram packet in the processRequest method. After receiving the packet, the source address and port are extracted from the packet, so that a reply packet can be constructed.

## send(DatagramPacket)

**ClassName**

DatagramSocket

**Purpose**

This method sends a datagram packet to a desired datagram socket.

**Syntax**

public void send(DatagramPacket packet)

**Parameters**

***packet***

The datagram packet that is to be sent.

**Description**

This method sends the specified packet to the destination address. The destination address and port number are contained within the packet. The underlying network will take the effort to deliver the packet to the destination. The delivery is not guaranteed, making the datagram communication unreliable.

**Imports**

*import java.net.DatagramSocket;*

**Returns**

None.

**Example**

In Listing 10-1, the server constructs a message, such as "This is message#1", and constructs a packet by specifying the destination address. In the processRequests method in messageServer class, the packet is sent to a client using the send method.

## DatagramPacket

**Purpose**

Use the DatagramPacket class to send messages when you develop applications based on connectionless protocol.

**Syntax**

public final class DatagramPacket extends Object

**Description**

An instance of this class forms a datagram packet in a client-server application, which is implemented using the connectionless protocol. It uses the UDP/IP protocol over the Internet. This is the class that should be instantiated to wrap up the data to be sent. The packet is self contained with the data and the destination address. It contains the data buffer, the packet length, and the destination InetAddress and port number. A packet is sent to a target using the send method of the DatagramSocket class. A packet is received using the receive method of the DatagramSocket class. When sending a packet, the Datagram(byte[], int, InetAddress, int) constructor is used to create the packet to be sent. It is created by specifying the destination address. When receiving a packet, the DatagramPacket object is constructed using the DatagramPacket(byte[], int) constructor. In either of the constructors, if the specified length is greater than the buffer length, an IllegalArgumentException is thrown. Figure 10-8 illustrates the inheritance diagram for the DatagramPacket class.

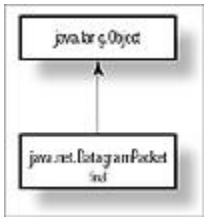


**Figure 10-8** Class diagram for DatagramPacket class

**PackageName**

*import java.net*

**Imports**

*import java.net.DatagramPacket;*

**Constructors**

public DatagramPacket(byte[] buffer, int *length*)

public DatagramPacket(byte[] buffer, int *length*, InetAddress *dest*, int *port*)

**Parameters**

***buffer***

The byte array containing the data.

***length***

The length of the data buffer.

***InetAddress***

IP address of the receiving datagram (where the packet is sent).

***port***

The port number of the receiving datagram.

**Example**

Refer to the processRequests method in the messageServer class in Listing 10-1. When a packet is received from the client, the size is specified as 256 and the packet is filled by using the receive method. Then the packet is constructed by specifying the destination address along with the data and sent to the client.

## getAddress()

**ClassName**

DatagramPacket

**Purpose**

Obtains the InetAddress from which the datagram packet was sent.

**Syntax**

public InetAddress getAddress()

**Parameters**

None.

**Description**

This method returns the InetAddress from which the packet was sent. This address is a part of the packet received. Typically on the server side, the IP address of the client which sent the packet is determined using this method.

**Imports**

*import java.net.DatagramPacket;*

**Returns**

The Internet address of the source; return type is InetAddress.

**See Also**

The class InetAddress in the *java.net* package

**Example**

In Listing 10-1, the InetAddress of the client datagram is obtained by using this method. This address is in turn used to construct the reply datagram packet to specify the destination.

## getData()

**ClassName**

DatagramPacket

**Purpose**

Obtains the data part of the received message.

**Syntax**

public byte[] getData()

**Parameters**

None.

**Description**

This method returns the data part of the datagram packet. It returns an array of bytes. If the message is expected to be a string, a String object is created by passing the byte array as a parameter for its constructor.

**Imports**

*import java.net.DatagramPacket;*

**Returns**

The data part of the packet; return type is byte[].

**Example**

In Listing 10-2, the message, received from the server, is retrieved from the packet using this method. A String object *msg* is constructed by passing, as a parameter, the byte array returned by this method. The message part of the received packet is printed to the screen.

## getLength()

**ClassName**

DatagramPacket

**Purpose**

Obtains the length of the packet.

**Syntax**

public int getLength()

**Parameters**

None.

**Description**

This method returns the length of the datagram packet. It is actually the size of the buffer containing the data.

**Imports**

*import java.net.DatagramPacket;*

**Returns**

The packet length; return type is int.

**Example**

In Listing 10-2, the length of the received packet is obtained from the packet using this method.

## getPort()

**ClassName**

DatagramPacket

**Purpose**

Obtains the port number from which the datagram packet was sent.

**Syntax**

public int getPort()

**Parameters**

None.

**Description**

This method returns the port number from which the packet was sent. This port number is a part of the packet received. Typically, on the server side, the port number of the client which sent the packet is determined using this method

**Imports**

*import java.net.DatagramPacket;*

**Returns**

The port number of the source; return type is int.

**Example**

In Listing 10-1, the port number of the client datagram is obtained by using this method. This number is, in turn, used to construct the reply datagram packet to specify the destination.

## SocketImpl

**Purpose**

An abstract socket implementation class that should be subclassed to provide actual implementation.

**Syntax**

public class SocketImpl extends Object

**Description**

By subclassing this class and implementing the methods, desired policies for socket implementation can be provided. Then, by using the factory to generate instances of this implementation class, server socket and client socket can be implemented. This class and its methods will not be normally used by programmers because the APIs for Socket and ServerSocket have a default

SocketImpl defined. Once a subclass of this class is defined, it should be generated by an implementation of SocketImplFactory interface. The factory of Socket and/or ServerSocket can be set to this factory interface, by using the setFactory method in those classes. Figure 10-9 illustrates the inheritance diagram for the SocketImpl class.

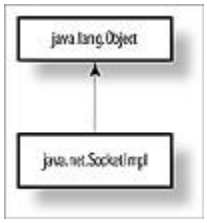


**Figure 10-9** Class diagram for SocketImpl class

**PackageName**

*java.net*

**Import**

*import java.net.SocketImpl;*

**Constructors**

public SocketImpl()

**Parameters**

None.

**Example**

Refer to the code example in setSocketImplFactory(SocketImplFactory) method in the class Socket.

## accept(SocketImpl)

**ClassName**

SocketImpl

**Purpose**

Make this server socket accept connection from a client socket with the specified SockImpl policies.

**Syntax**

protected abstract void accept(SocketImpl s_imp) throws IOException

**Parameters**

***s_imp***

SocketImpl of a client socket trying to connect to this server.

**Description**

This method should be defined in the subclass of the SocketImpl class. It makes sure that the server socket will accept connection only from a client socket with specified characteristics. This will help to build firewalls and restrict access by defining the client sockets that can connect to a server socket.

**Imports**

*import java.net.SocketImpl;*

**Returns**

None.

**See Also**

The class ServerSocket; interface SocketImplFactory

**Example**

The following example illustrates a sample implementation of this accept(SocketImpl) method.

```
import java.net.ServerSocket;
import java.net.SocketImpl;
import java.net.SocketImplFactory;

/* subclass the SocketImpl class to provide actual implementation of
socket policies */

class MySockImpl extends SocketImpl {

 /* Implement  the methods of SocketImpl class to adhere by the desired
client socket policies */

/* defining accept method */

protected synchronized void accept(SocketImpl s_imp) throws
IOException {
             socketAcceptMethod(s_imp);/*call a native method */
     }

private native void socketAcceptMethod(SocketImpl s_imp)
      throws IOException;

}

/* define a factory to generate instances of MySockImpl class*/
class MyFac implements SocketImplFactory {

      SocketImpl createSocketImpl() {
              /* return an instance of MySockImpl */
      }
}

/* A class using server socket and defining its behavior */
public class Server {

  public static void main(String args[]) {
      SocketImplFactory fac = new MyFac();
      Socket.setSocketImplFactory(MyFac);

/* Any new instances of ServerSocket created hereafter will implement
   the policies defined by*/
   MySockImplclass and the accept method will follow the policies
   defined in the native method,*/
   socketAcceptMethod() */

             } /* end of main() */
} /* end of class Serv */
```

## available()

**ClassName**

SocketImpl

**Purpose**

Gets the number of bytes that can be read without blocking.

**Syntax**

protected abstract int available() throws IOException

**Parameters**

None.

**Description**

This method should be defined in the subclass of the SocketImpl class. This returns the number of bytes a socket can read without blocking. This method can be used to read from the input stream of the socket.

**Imports**

*import java.net.SocketImpl;*

**Returns**

The return type of this method is int.

## bind(InetAddress, int)

**ClassName**

SocketImpl

**Purpose**

Binds the server socket to the specified port on a host specified by the InetAddress.

**Syntax**

protected abstract void bind(InetAddress inet, int port) throws IOException

**Parameters**

***inet***

InetAddress of the host to which the socket should be bound.

***port***

Port number on the above host to bind the socket to.

**Description**

This method should be defined in the subclass of the SocketImpl class. After creating a socket instance, this method is invoked to bind the server socket to the specified port on a host.

**Imports**

*import java.net.SocketImpl;*

**Returns**

None.

**See Also**

The class InetAddress; interface SocketImplFactory; class ServerSocket

**Example**

This method is used by ServerSocket class to bind to a specified port on a given host. You should write your own native method to implement this method by subclassing the SocketImpl class.

## close()

**ClassName**

SocketImpl

**Purpose**

Closes the socket connection.

**Syntax**

protected abstract void close() throws IOException

**Parameters**

None.

**Description**

This method should be defined in the subclass of the SocketImpl class. This method is invoked when Socket.close() or ServerSocket.close() is called.

**Imports**

*import java.net.SocketImpl;*

**Returns**

None.

**See Also**

The class ServerSocket; class Socket; interface SocketImplFactory

**Example**

Here is an example usage and definition of this method in a subclass of SocketImpl.

```
import java.net.Socket;
import java.net.SocketImpl;
import java.net.SocketImplFactory;

/* subclass the SocketImpl class to provide actual implementation of
socket policies */

class MySockImpl extends SocketImpl {

 /* Implement  the methods of SocketImpl class to adhere by the desired
client socket policies */

/* defining close method */

protected synchronized void close(SocketImpl s_imp) throws
IOException {
              socketCloseMethod();/*call a native method */
      }
private native void socketCloseMethod() throws IOException;

}

/* define a factory to generate instances of MySockImpl class*/

class MyFac implements SocketImplFactory {
```

```
      SocketImpl createSocketImpl() {
           /* return an instance of MySockImpl */
      }
}
```

## connect(String, int), connect(InetAddress, int)

**ClassName**

SocketImpl

**Purpose**

Use these methods to define the connect method of a client socket if you are providing your own SocketImpl.

**Syntax**

protected void connect(String name, int port) throws IOException
protected void connect(InetAddress addr, int port) throws IOException

**Parameters**

***name***

Name of the host to which the socket is connected.

***addr***

Address of the host to which the socket is connected.

***port***

Port number on the host, to which the socket connection is to be established.

**Description**

These methods should be defined in the subclass of the SocketImpl class. Given the port number and the name or InetAddress of the host machine where a server is available, these methods provide the implementation for the connect() method in Socket class. This method is invoked during construction of a Socket instance, after create() method is invoked.

**Imports**

*import java.net.SocketImpl;*

**Returns**

None.

**See Also**

The class Socket; interface SocketImplFactory

**Example**

The following code illustrates an example implementation outline for these methods.

```
import java.net.Socket;
import java.net.SocketImpl;
import java.net.SocketImplFactory;

/* subclass the SocketImpl class to provide actual implementation of
socket policies */

class MySockImpl extends SocketImpl {

 /* Implement  the methods of SocketImpl class to adhere by the desired
socket policies */
```

```
/* defining connect method */

protected void connect(String host, int port) throws
IOException {
            socketConnectMethod(host, port);
                /*call a native method */
        }

private native void socketConnectMethod(String host, int port)
throws IOException;

protected void connect(InetAddress addr, int port) throws
IOException {
            socketConnectMethod(addr, port);
                /*call a native method */
        }

private native void socketConnectMethod(InetAddress addr, int port)
throws IOException;

}

/* define a factory to generate instances of MySockImpl class*/

class MyFac implements SocketImplFactory {

      SocketImpl createSocketImpl() {
           /* return an instance of MySockImpl */
      }
}
```

## create(*boolean*)

**ClassName**

SocketImpl

**Purpose**

Creates either a stream socket or a datagram socket.

**Syntax**

protected abstract void create(boolean is_stream) throws IOException

**Parameters**

***is_stream***

The value of this variable should be set to true if you need a stream socket; if you need a datagram socket, this should be set to false.

**Description**

This method should be defined in the subclass of the SocketImpl class. Depending on the value of the boolean input, either a stream socket or a datagram socket is created. This is used while constructing instances of both Socket and ServerSocket. The default socket created by using the SocketImpl implemented is a stream socket.

**Imports**

*import java.net.SocketImpl;*

**Returns**

None.

**See Also**

class ServerSocket; class Socket; interface SocketImplFactory

**Example**

The following example illustrates a sample implementation outline for this create(boolean) method.

```
import java.net.Socket;
import java.net.SocketImpl;
import java.net.SocketImplFactory;

/* subclass the SocketImpl class to provide actual implementation of
socket policies */

class MySockImpl extends SocketImpl {

 /* Implement  the methods of SocketImpl class to adhere by the desired
client socket policies */

/* defining create method */

protected synchronized void create(boolean is_stream) throws
IOException
{
             socketCreateMethod(is_stream);/*call a native method */
     }

private native void socketCreateMethod(boolean is_stream)
      throws IOException;

}

/* define a factory to generate instances of MySockImpl class    */
class MyFac implements SocketImplFactory {

      SocketImpl createSocketImpl() {
            /* return an instance of MySockImpl */
      }
}
```

## getFileDescriptor()

**ClassName**

SocketImpl

**Purpose**

The file descriptor of the socket is returned.

**Syntax**

protected FileDescriptor getFileDescriptor() throws IOException

**Parameters**

None.

**Description**

Each socket has a file descriptor associated with it. It is the way in which the socket is implemented. This method returns a handle to the file descriptor.

**Imports**

*import java.net.SocketImpl;*

**Returns**

The file descriptor associated with the socket.

## getInetAddress()

**ClassName**

SocketImpl

**Purpose**

Returns the InetAddress of the machine to which the socket is connected.

**Syntax**

protected InetAddress getInetAddress()

**Parameters**

None.

**Description**

A socket connects to a machine and performs its task of communicating with other hosts. The InetAddress associated with the server's host is obtained using this method.

**Imports**

*import java.net.SocketImpl;*

**Returns**

InetAddress representation of the machine to which the socket is connected.

**See Also**

The class ServerSocket; class Socket; interface SocketImplFactor

## getInputStream()

**ClassName**

SocketImpl

**Purpose**

An InputStream for this socket is obtained.

**Syntax**

protected abstract InputStream getInputStream() throws IOException

**Parameters**

None.

**Description**

An InputStream for the socket is returned to communicate with the paired endpoint of the point-to-point communication. This method should be defined in the subclass of the SocketImpl class.

**Imports**

*import java.net.SocketImpl;*
*import java.io.InputStream;*

**Returns**

An instance of InputStream class.

**See Also**

The class ServerSocket; class Socket; interface SocketImplFactory

**Example**

The following example illustrates a sample implementation outline for this getInputStream() method.

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

class MySockImpl extends SocketImpl {

     /* defining getInputStream method */

protected synchronized void getInputStream() throws IOException {
     socketgetInputStreamMethod();/*call a native method */
}

private native void socketgetInputStreamMethod()
          throws IOException;

}
```

## getLocalPort()

**ClassName**

SocketImpl

**Purpose**

Returns the local port to which this socket is bound.

**Syntax**

protected int getLocalPort()

**Parameters**

None.

**Description**

Every socket has a corresponding local port to which it has connected. For example, in case of a client, though it has connected to a server socket for service, it has an associated local port to which the server connects and sends a reply.

**Imports**

*import java.net.SocketImpl;*

**Returns**

Port number of the local port.

**See Also**

The class ServerSocket; class Socket; interface SocketImplFactory

## getOutputStream()

**ClassName**

SocketImpl

**Purpose**

An OutputStream for this socket is obtained.

**Syntax**

protected abstract InputStream getOutputStream() throws IOException

**Parameters**

None.

**Description**

An OutputStream for the socket is returned to communicate with other sockets. This method should be defined in the subclass of the SocketImpl class.

**Imports**

*import java.net.SocketImpl;*

**Returns**

An instance of OutputStream class.

**See Also**

The class ServerSocket; class Socket; interface SocketImplFactory

**Example**

The following example illustrates a sample implementation outline for this getOutputStream method.

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

class MySockImpl extends SocketImpl {

   /* defining getOutputStream method */

   protected synchronized void getOutputStream() throws    IOException
{
           socketgetOutputStreamMethod();/*call a native method */
   }

   private native void socketgetOutputStreamMethod()
           throws IOException;
   }
```

## getPort()

**ClassName**

SocketImpl

**Purpose**

Returns the number representing the server's port to which the client connects.

**Syntax**

protected int getPort()

**Parameters**

None.

**Description**

A client socket connects to a server at a specified port number, on which the server listens for connections. This method returns the number of this port.

**Imports**

*import java.net.SocketImpl;*

**Returns**

The number of the port on which the server is listening; return type is integer.

**See Also**

class Socket; interface SocketImplFactory

## listen(*int*)

**ClassName**

SocketImpl

**Purpose**

The server socket listens for connection from a client for a given amount of time.

**Syntax**

protected abstract void listen(int time_count) throws IOException

**Parameters**

***time_count***

The amount of time server will listen for connection from clients.

**Description**

This method should be defined in the subclass of the SocketImpl class. This method is invoked during construction of a server socket. After binding to specified host and port, the server (an instance of ServerSocket) listens for connection from clients for a specified amount of time.

**Imports**

*import java.net.SocketImpl;*

**Returns**

None.

**See Also**

The class ServerSocket; interface SocketImplFactory

**Example**

A sample outline is given below to illustrate the definition of the listen(int) method in a subclass of SocketImpl class.

```
   import java.net.*;

class MySockImpl extends SocketImpl {


   /* defining listen method */

   protected synchronized void listen(int time) throws    IOException {
         socketListenMethod(time);/*call a native method */
   }

   private native void socketListenMethod(SocketImpl s_imp)
         throws IOException;

}

   /* define a factory to generate instances of MySockImpl class*/
   class MyFac implements SocketImplFactory {
```

```
        SocketImpl createSocketImpl() {
              /* return an instance of MySockImpl */
        }
   }

   /* A class using server socket and defining its behavior */
   public class Server {

   public static void main(String args[]) {
         SocketImplFactory fac = new MyFac();
         Socket.setSocketImplFactory(fac);

   /* Any new instances of ServerSocket created hereafter   will
       implement the policies defined by MySockImplclass and the
accept()
       method will follow the policies defined in the native method,
       socketListenMethod() */

                } /* end of main() */
   } /* end of class Server */
```

## toString()

**ClassName**

SocketImpl

**Purpose**

To get a string form of the socket implementation details.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method obtains the port number, file descriptor, and address to which socket is connected, as a String. The file descriptor, address, and port number are concatenated into string form within a String object.

**Imports**

*import java.net.SocketImpl;*

**Returns**

The return type of the method is a String containing the port number, file descriptor, and address of the machine which the server is bound to or to which the client is connected.

**See Also**

The class ServerSocket; class Socket

**Example**

The following example gives an outline of implementing this method.

```
import java.net.*;

class MySockImpl extends SocketImpl {

        /* defining toString method */
```

```
protected synchronized String toString() throws
IOException {
        sockettoStringMethod();/*call a native method */
}

private native void sockettoStringMethod()
        throws IOException;

}
```

## SocketImplFactory

**Purpose**

An interface to define a factory for actual SocketImpl instances.

**Syntax**

public interface SocketImplFactory extends Object

**Description**

This interface can be used by various socket classes to implement their policies. By subclassing the SocketImpl class and implementing its methods, desired policies for socket implementation can be provided. Then, by using SocketImplFactory to generate instances of the defined SocketImpl class, a server socket and client socket can be implemented. The factory of Socket and ServerSocket should be set to this factory interface, using the setFactory method in those classes. The factory of Socket or ServerSocket can be specified only once. This interface should be defined only if SocketImpl is subclassed.

**PackageName**

*java.net*

**Imports**

*import java.net.SocketImplFactory;*

**Constructors**

None.

**Parameters**

None.

**Example**

Refer to the example in class SocketImpl.

## createSocketImpl()

**Interface**

SocketImplFactory

**Purpose**

To generate an instance of SocketImpl that defines the actual socket implementation policies.

**Syntax**

public abstract SocketImpl createSocketImpl()

**Parameters**

None.

**Description**

This method should be used to generate instances of a user-defined SocketImpl subclass. This subclass will implement the socket policies desired by the user. This will be invoked by the Socket or ServerSocket constructors to define their SocketImpl member.

**Imports**

*import java.net.SocketImplFactory;*

**Returns**

This method returns an instance of a subclass of SocketImpl.

**See Also**

The class SocketImpl

**Example**

Refer to the example in class SocketImpl.

## The Network and Sockets Project: A Client-Server Rendezvous Applet

In the *Client-Server Rendezvous* applet, a client will contact an existing server and obtain details. You should provide three buttons: clientinfo, serverinfo, and fileinfo. On clicking the clientinfo button, the client will print out its details, the host it is running on, and the port on which it is connected. It should be able to gather the information about the server and print the details when the serverinfo button is pressed. When the fileinfo button is pressed, the client will send a request to the server to obtain the contents of a file residing on the server's side.

The applet should also be a stand-alone application, so that you can run it from a Java environment using the Java interpreter or it can also be launched from a Web browser, supporting Java. The scenario for this applet is illustrated in Figure 10-10. The steps involved are as follows:

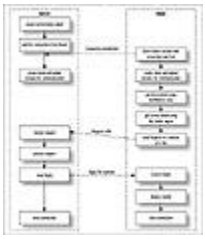

**Figure 10-10** Sequence of actions in the Client-Server Rendezvous

**1.** The server connects to a port on the host, to which it is initialized, and listens for connections from clients.
**2.** A client will connect to the server by specifying the hostname and the port on which the server is listening.
**3.** The client retrieves the details about itself and displays the information.
**4.** The client obtains the information about the server and displays it.
**5.** The client sends a request to the server asking for the contents of a file on the server's side.
**6.** The server, on receiving the request, opens the relevant file, if it exists, and sends the contents to the client over the socket streams.

**7.** The client receives the file contents from the server and displays them.
**8.** Once the necessary processing is completed, the streams and sockets are closed appropriately.

You must now be able to implement the applet using the APIs described in this chapter. As you know, any applet for a given specification can be implemented in different ways. One such implementation is provided here. In this implementation, Threads are used on the server's side to process multiple clients. In the given applet, the Server is started on a given machine. The Client can either be run as a stand-alone application using the Java interpreter or launched from a Web browser supporting Java. In either case, the server hostname and port number are passed as arguments to the executable.

## Building Your Applet

**1.** First create a Server class which should accept connection from clients. Make it a public class and the filename should be Server.java. This class should contain the following members as listed in Table 10-2.

**Table 10-2** Members of Server class

| Modifier | Type | Variable-name | Purpose |
|---|---|---|---|
| static | ServerSocket | *ServSock* | An instance of ServerSocket is created to enable the server to accept connections from clients. |
| static | Socket | *theSocket* | An instance of Socket that will be created on the Server side when ServerSock is instantiated. |
| static | int | *port* | Port number on which the server connects and creates ServerSock |
| static | ServerThread | *client[]* | An array of ServerThread objects which will act as servers for every client that connects to the server, so that there is one ServerThread for one client. |

After the addition of these members, your file Server.java should contain the following:

```
// import the necessary classes relevant to the class Server

import java.net.Socket;
import java.net.ServerSocket;
import java.io.*;
/**
```

```
        Class name: Server
*/

public class Server {

        private static Socket theSocket;
        private static ServerSocket ServSock;
        private static ServerThread client[] = new ServerThread[10];
        private static int port = 3001;
}
```

**2.** Having created the class, now define the main method for the class. Create an instance of ServerSocket after specifying the port number. Then the server waits in a loop for connections from clients. This is done using the accept() method of ServerSocket. After accepting connection from a client, a server thread is spawned for each client. This enables exclusive service to the client by a corresponding server Thread. The start() method of Thread will start the servicing of the server Thread to the client. Adding these functions to your Server class will make the Server.java file contain the following code:

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.*;

/**
       Class name: Server
       This class is run as a background process.
        eg.,   % java Server &

       It accepts connection from clients and spawns a server
Thread
       for every such connection and goes back to listen for more
       connections. Maximum of two clients can be active at a
given
       instant. Processing of client requests is done by the
       corresponding server thread.
*/

public class Server {

       static Socket theSocket;
       static ServerSocket ServSock;
       static ServerThread client[] = new ServerThread[10];

       DataInputStream datain;
       DataOutputStream dataout;
       static int port = 3001;

       public static void main(String args[]) {
         int g=0;
         try {
              ServSock = new ServerSocket(port);
              System.out.println("Server started");
              while (true) {
                      theSocket = ServSock.accept();
                      for (g=0;g<10;g++)
                      if ((client[g] == null) || (!client[g].
```

```
                        isAlive()))
                                        break;

                        if (g<10) {
                                    client[g] = new ServerThread
                                    (theSocket,g);
                                    client[g].start();
                        }
                                    else
                                    System.out.println("Rejected a
                                    connection");
                }
                } catch (IOException ioE){
                  System.out.println(" Server error ");
              }
            }
}
```

**3.** Now you have the necessary code for a Server to run. Next, you should write the necessary code for implementing the ServerThread the Server spawns for every Client. Create a public class ServerThread. It should be a subclass of Thread class in Java and should contain the members specified in Table 10-3.

**Table 10-3** Members of ServerThread class

| Modifier | Type | Variable name | Purpose |
|---|---|---|---|
| | Socket | *mySocket* | This Socket object will be the socket instance on Server's side for a connected Client. This is passed on to this ServerThread using the constructor. |
| | int | *myId* | Every ServerThread has an Id associated with it, which also identifies the Client. There can be only one ServerThread with a given Id, at a given instance. |
| | DataInputStream | *datain* | DataInputStream associated with every ServerThread (which will be the Client's OutputStream) to receive messages from the Client. |
| | DataOutputStream | *dataout* | DataOutputStream associated with every ServerThread (which will be the Client's InputStream) to send messages to the Client. |

With these data members and the constructor for the ServerThread class, your ServerThread.java should now contain the following:

```
import java.net.Socket;
import java.io.*;
import java.awt.*;
/**
       Class name: ServerThread

       This class subclasses the Thread class. This thread is spawned
by
       the Server object, that accepts connection from clients.
       Each ServerThread services requests from one client assigned by
       the Server
*/

public class ServerThread extends Thread {

       private Socket mySocket;
       private DataInputStream datain;
       private DataOutputStream dataout;
       private int myId;

       public ServerThread(Socket m, int Id) throws IOException {

          mySocket = m;
          myId = Id;

          datain = new DataInputStream(new
BufferedInputStream(mySocket.getInputStream()));
          dataout = new DataOutputStream(new
BufferedOutputStream(mySocket.getOutputStream()));
        }
}
```

**4.** Now you should override the run() method of Thread class in ServerThread class. Assuming you have a function processRequests() in this class, the run method will call the processRequests() while there are more requests from the Client. In the meantime, after processing every request, the Thread should yield to the other ServerThreads to process their respective Client requests. After all the requests are processed, you should close the sockets and streams that are open. Enter the following method into the ServerThread class.

```
       /**
         The "run" method of Thread class being implemented here
       */

      public void run() {
       try {
        while (processRequests())
         yield(); // yield to other threads too!

         CleanUp();
        } catch (IOException E){
              System.out.println("Error in processing request");
        }
               }
```

```
  /** clean up when done */

void  CleanUp() {
          try {
              datain.close();
              dataout.close();
              mySocket.close();
            catch (IOException io) {
                      printOut(" IOEXception!! ");
                      printOut(io.getMessage());
         }
      }
```

**5.** Let retrieving the contents of a requested file be a service provided by the ServerThread. When the ServerThread receives a message "File" from a Client through the DataInputStream, it understands that the Client is requesting a file to be retrieved and it expects another message from the Client indicating the name of the file to be retrieved. The ServerThread then reads the file and sends its contents using the DataOutputStream. If the message is "Bye", the ServerThread understands that the Client intends to close the session and so the method returns false. This makes the ServerThread's run() method terminate and so the ServerThread gets disposed. To achieve the described effect, include the following processRequests() method, whose return type is boolean. This method assumes the existence of a GetFile() method in this class.

```
/** This method processes client requests */

      private boolean processRequests() throws IOException {
         try {
               String req = datain.readUTF();
               if (req.equals("File")) {
                      String file = datain.readUTF();
                      GetFile(file);
                      return true;
               }
              else if (req.equals("Bye"))
                      return false;
              else {
                   System.out.println("Unknown service
requested");
                   return false;
              }
         } catch(IOException ioe) {
              System.out.println("Error in input from Client");
              return false;
         }
     }
```

**6.** As a final part of our ServerThread class, you should now implement the GetFile() method. Given a filename, this method will first check to see if the file exists and if it does, whether it is readable. Then using the file, the method creates a DataInputStream by passing the FileInputStream as a parameter. Next the method reads the file line-by-line and sends the line contents to the Client using the DataOutputStream object named dataout. It follows the file contents with an

EndOfFile message to the Client. Add the following code into the ServerThread class.

```
/**
        Method that retrieves contents of a file the client is
        interested in. It uses FileInputStream to achieve the
same */


  private  void GetFile(String file_name) {
          //buffer to get all the lines in the file
      StringBuffer buff = new StringBuffer();

      File f = new File(file_name);
      boolean b = (f.exists() || f.canRead());
      if (!b)
        printOut(" File either doesn't exist or is unreadable");
      try {
           DataInputStream f_in = new DataInputStream(new
                                       BufferedInputStream ( new

FileInputStream(file_name)));

          while(f_in.available() !=0) {
                  String line = f_in.readLine();
                  buff.append(line +"\n");
                  dataout.writeUTF(line);
                  dataout.flush();
          }
          dataout.writeUTF("EndOfFile");
          dataout.flush();
    } catch (IOException ioE) {
          System.out.println("Error in handling file");
    }
          }
```

**7.** The Client class is the next one to be created. Client is the class that will be launched as an applet from the Web browser. So it extends the applet and in this implementation, implements the Runnable interface. It acts as a client requesting service from an existing Server. According to the specification, this should also run as a stand-alone application. It has instances of Socket, DataInputStream, DataOutputStream, and Thread as its members. It should also implement user interface with three buttons: clientinfo, serverinfo and fileinfo. Enter the following code in the file Client.java.

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

/**
        Classname: Client
        This is a class that creates a client socket, Socket, and
        retrieves information about
                 client Host or
                 server Host or
                 requests a File from the server
```

```
        Illustrates the usage of APIs in the classes
                java.io.InetAddress
                java.io.Socket
**/


public class Client extends Applet  {
        private Socket sock;
        private DataInputStream datain;
        private DataOutputStream dataout;
        private String InpStr[];
        private int count;

        String servHost;
        int servPort;

        boolean standalone;
}
```

**8.** You should override the init() method of Applet class by creating a Panel with three buttons. Then the Client connects to an existing Server at a specified host and port. It also obtains the input and output streams. If the client is an applet, it can also obtain some of the parameters from the HTML file. The following init() method implements these specifications. Include this method in the Client class.

```
public void init() {
        resize(600,400);
        InpStr = new String[100];

        makePanel(); // create the panel with buttons

      if (!standalone) {
      // initialize parameters using the values from the html doc

         String at = getParameter("servHost");
         servHost = (at != null) ? at : "serval.cat.syr.edu";
         at = getParameter("servPort");
       servPort=(at != null)?Integer.valueOf(at).intValue():3001;
      }

      makeConnection();  // establish connection

 }

private void makePanel() {

        /*to create buttons*/
        Panel p = new Panel();
        p.setLayout(new FlowLayout());
        add("South", p);
        p.add(new Button("clientinfo"));
        p.add(new Button("serverinfo"));
        p.add(new Button("fileinfo"));
}

private void makeConnection() {
        /* Open socket to the server and set up the streams */
  try {
```

```
       InetAddress addr = InetAddress.getByName(servHost);
       sock = new Socket(addr,servPort); //create a client socket
    //Obtain input and output stream to communicate with the
server
       datain = new DataInputStream(new
                   BufferedInputStream(sock.getInputStream()));
       dataout = new DataOutputStream(new
                   BufferedOutputStream(sock.getOutputStream()));

  } catch (IOException E) {
         System.out.println(" IOException in client!!");
        System.out.println(E.getMessage());
   }
}
```

**9.** To implement this as a stand-alone application you need to write a method main(). The following code listing implements this method, which obtains the server name and port number from the command line. An instance of Client is created and a method myinit() is invoked to pass the parameters to the Client object. Then the Client Thread is started and a Frame is initialized to contain the three buttons to be created. Enter the following code in the Client class to extend it as a stand-alone application.

```
public static void main(String args[]) throws IOException {

   Frame f = new Frame("Client-Server Rendezvous");
           // obtain the port number from second parameter on
command
              line.
   int port = (new Integer(args[1])).intValue();
                                    // convert it to an integer
from
                                       its string value
   Client Clnt = new Client();     // create an instance of Client
   Clnt.myinit(args[0],port);      // initialize and start it
   Clnt.start();

   f.add("Center", Clnt);          // create the frame
   f.resize(600,800);
   f.show();
}

public void myinit(String Host, int port) {
     standalone = true;
     servHost = Host;
     servPort = port;
     init();
}
```

**10.** Now that you have Buttons in the Panel, you have to override the action() method so that appropriate action is taken when a button is pressed. The following code achieves this. Enter the code in the class Client. Also include two variables, *InpStr* and *count. InpStr* is an array of String that will contain the string to be printed on the canvas. The variable *count* will keep track of the lines printed out to the canvas.

```
String InpStr[];
int count;
```

```
 /** Action for Button event */
 public boolean action(Event evt, Object arg) {
 if(evt.target instanceof Button) {   // if a Button is pressed
      count=0;
      if("clientinfo".equals(arg)) {   // if clientinfo button
                                          is selected
            int port_num = sock.getPort();
            printOut(" Client has connected to a Server listening
at
            the port number " + port_num );
            ClientInfo();
            printOut("\n\n");
      }
      else if("serverinfo".equals(arg)) {   // if serverinfo is
                                              requested
             ServerInfo();
             printOut("\n\n");
          }
      else if("fileinfo".equals(arg)) {   // if  fileinfo is
                                             requested
             try {
                      // you are requesting the contents of file
by
                         name /etc/motd
             String file_name = new String("/etc/motd");
             MakeRequests(file_name);
             } catch (ArrayIndexOutOfBoundsException a) {
             printOut(" For accessing remote file \n \t \tUsage:
             java Client <filename> ");
             }
             printOut("\n\n"); // a pretty print method available
             within this class
      }
    }
   return true;
 }
```

**11.** Information about the Client is to be retrieved when the clientinfo button is pressed. Including the following method in Client class will make this happen. If the InetAddress of the local host is made available, then more details can be obtained from the InetAddress instance that will reflect the client machine information.

```
/** Method to obtain information about the client host */

public void ClientInfo() {
   InetAddress c_inet;
   String c_name;

    try {
        c_inet = InetAddress.getLocalHost();
        // InetAddress of the local host
        c_name = c_inet.getHostName();
        // get the host name of the client
        printOut(" Client Host Details ");
        printOut(" HostName : " + c_name);
```

```
        // get the string form of the inet address and extract
the IP
        address part of it
        String c_str = c_inet.toString();
        int index = c_str.indexOf('/');
        String c_ipaddr = c_str.substring(index+1);
        printOut(" IP Address : " + c_ipaddr );
   } catch (IOException ioE);
}
```

**12.** Server details can be obtained from the server's InetAddress in a similar manner as from the Client's. To get the InetAddress of the server, the Socket instance is used. The getInetAddress() method of Socket class is used. Include the following code in the Client class.

```
/** Method to obtain information about the server */

public void ServerInfo() {
        InetAddress s_inet = sock.getInetAddress();
        String s_name = s_inet.getHostName();
        printOut(" Server Host Details ");
        printOut(" HostName : " + s_name);

        String s_str = s_inet.toString();
        int index = s_str.indexOf('/');
        String s_ipaddr = s_str.substring(index+1);
        printOut(" IP Address : " + s_ipaddr );
}
```

**13.** The following MakeRequests() method is used to send the file name to the server and request the contents of the file. The client then reads the reply from the server and prints it on the screen until the end of file is reached. The datain and dataout members of type DataInputStream and DataOutputStream are used by the Client to communicate with the Server.

```
/** Client makes its requests to the server by sending messages
    over sockets
                Remote File retrieval is done in this method
*/

public void MakeRequests(String fil_n) {

/* Get a file */


    printOut(" File requested by the client: " + fil_n);
    printOut("\n\n");
    printOut(" The file contains the following: ");
    printOut("\n\n");
     try {
        dataout.writeUTF("File");   // inform the server that you
are
                                       requesting a File
        dataout.flush();            // send the filename to the
server
        dataout.writeUTF(fil_n);
        dataout.flush();            // always use flush() after
using
                                       write() method of
```

```
                                        // OutputStream
        String file_contents = datain.readUTF();
        while (!file_contents.equals("EndOfFile")) {
                // get the contents of file line by line and
print
                   them
                   printOut(file_contents);
                   file_contents = datain.readUTF();
                }
        dataout.writeUTF("Bye");         // transaction is
complete
        dataout.flush();
      }   catch (IOException ioE){
                              // catch the i/o exception
                      System.out.println(" Oops! file prob");};
   }
```

**14.** Include the following methods in the class Client. To print the strings on the canvas in an orderly manner, write the printOut() method, which is used by other methods to print on the canvas. The paint() method is overridden here to write to the exact locations on the canvas.

```
/** Method to print the obtained strings to output stream
*/

public void printOut(String str) {
        InpStr[count] = new String(str);
        count++;
        repaint();
         try {
               mythread.sleep(500);
        } catch (InterruptedException ie){};

}

/**
 * Paint it.
 */

 public void paint(Graphics g) {
   Dimension d = size();
   g.setColor(Color.black); // write the contents in black

   // write the contents line by line as string
   //for simplicity there are only maximum 60 lines allowed
   for (int y=60, i = 0; i <count; i++) {
     y +=20; // between each line leave 20 pixels gap

     // draw the string at 40th column and specified line 'y'
     g.drawString(InpStr[i],40,y);
   }
}
```

**15.** You should take necessary care to close any open files, streams, or sockets. This can be done in the stop() method of the Applet, which is called when the Applet is terminated.

```
/* Applet stop */
public void stop() {
```

```
        System.out.println(" inside Client.stop() ");
         if (mythread != null) {
              mythread.stop();
              mythread=null;
        /* additional cleanup (closings) */
             try {
                dataout.close();
                datain.close();
                sock.close();
             } catch (IOException E);
       }
}
```

**16.** The above three files are compiled using javac. The server is executed using the Java interpreter. Use

```
java Server
```

at the command prompt to run the Server.

**17.** The Client applet can be launched from the Web using the following HTML file, csr.html.

```
<title> Client-Server Rendezvous </title>
<hr>
<applet code=Client.class width=600 height=400>
<param name=servHost value="serval.cat.syr.edu">
<param name=servPort value=3001>
</applet>
<hr>
```

The applet, when launched, using the command appletviewer csr.html, will create a Panel that will appear as in Figure 10-11. When you press any of the three keys, appropriate action is taken and details are printed on the canvas. This applet implements the Client-Server Rendezvous and exchange of information between the Client and the Server. This illustrates a typical client-server application.

**Figure 10-11** Client-Server Rendezvous applet in action

## How It Works

The project developed in this chapter is a client-server application. The server is a stand-alone application. First, start the server on the host you want to run the server. In the code, we had the host named as serval.cat.syr.edu. If the host you are running your server on is foo.bar.usa, change the string serval.cat.syr.edu in the csr.html file to foo.bar.usa. After starting the server, run the client applet. When the applet comes up, it displays a window with three buttons: clientinfo, serverinfo, fileinfo. If you click the clientinfo button, the details of the host, on which the client applet is executed, is displayed on the canvas. If you click the serverinfo button, the details of the server host, host on which the

server is running, is displayed on the window. Whereas if you click on the fileinfo button, the contents of the /etc/motd file (in case of Unix systems) is displayed on the screen. If you are interested in any other file, change the filename in the code to the desired filename. As an exercise, change the code such that the filename is passed as a parameter from the command line prompt (in case of stand-alone application) or from the HTML file (in case it's launched from a browser). This project gives you a good start in developing networking applications in Java. Happy networking!

# Chapter 11
# Handling URLs And Networking Exceptions

If you are developing Java applications for the World Wide Web, this chapter will help you start writing Java applets that will navigate the Web. It introduces you to some of the basic concepts of the World Wide Web and explains, in detail, the Java classes that help you tap into the resources on the Web from within a Java application. In addition, it describes the exception signals that are thrown when an error is detected while connecting to or using a resource available over the network. The chapter project illustrates how you can use the Java classes to access objects specified by Uniform Resource Locator addresses. This will help you get started with writing your own Java programs that interact with the Internet.

## URLs, Protocols, and MIME

As its name indicates, the World Wide Web is a global network of computers. Web clients request and receive information from Web servers. Just as in postal addresses in different countries, the location of information may vary from one computer to another. If each computer had its own addressing scheme for retrieving its data, then the process of transparently accessing this information would become very difficult. The architects of the Web formulated an addressing scheme that could be used by one and all to serve and access information on the Web. Information on this network is accessible using an address specification called a URL. URL stands for Uniform Resource Locator. It is a structured address that uniquely identifies a resource (be it a document, an image, or whatever) on the World Wide Web.

A complete URL consists of a protocol specifier followed by a string, whose format depends on the protocol specification. Many protocols are supported on the World Wide Web, the most popular being the HTTP or HyperText Transfer Protocol. Other popular protocols are news (to read Usenet newsgroups), gopher (the Gopher protocol) and many more. The client-server interaction is markedly different among the various protocols. The basic syntax of a HTTP URL is as follows:

http://hostname[:portnumber]/directory/filename

The http denotes the protocol type. Colons and slashes (/) are used as delimiters. The hostname field is used to specify an Internet hostname (e.g., www.syr.edu). The portnumber is an optional field that is used to specify the port on the target host, at which an http server is running. If it is omitted then the default port for the protocol is used. The directory and file name fields are used to specify the path name of the document that is to be retrieved. This path name is relative to the directory that the http server makes public. Figure 11-1 looks at an example of an HTTP Uniform Resource Locator in detail.

**Figure 11-1** Anatomy of a HyperText Transfer Protocol (HTTP) URL

World Wide Web browsers have become very popular. Information that is out there on the Internet is just a click of the mouse button away. How does the browser distinguish between image files, text files, audio files, and the many other file formats that exist? Part of the protocol between the client and the server involves sending some header or context information about the data that is being sent by the server. The Multipurpose Internet Mail Extensions (MIME) format specifies this context information. A MIME type is of the format:

type/subtype.

Using this information, the client (browser) can identify the type of file that it retrieved. The major types supported on the World Wide Web are: text, image, audio, video, and application. Even among these major types of files there are different formats. For example, two popular data formats for image files are GIF (Graphics Interchange Format) and TIFF (Tagged Image File Format). The subtype of the MIME-type specifies the exact format of the file. This table lists some sample content types and their corresponding MIME-types:

| Content Type | MIME-Type/Subtype |
|---|---|
| GIF image | image/gif |
| TIFF image | image/tiff |
| Plain text file | text/plain |
| HTML file | text/html |
| Audio file (AU format) | audio/basic |

## Java and the World Wide Web

Java provides many classes that can be used to write applications that access resources on the World Wide Web. The URL class encapsulates the concept of a Uniform Resource Locator. The URL class and the Applet class will satisfy most of the needs of Java Web applications. The designers of the Java Development Kit included a set of extensible classes that are very useful for programmers who are developing Web browsers and for those involved in developing protocols to be used on the Web.

Each protocol has its own set of specifications for client-server interaction. If you want to write your own protocol handler in Java, you must extend and implement a number of classes. The first of these is the URLStreamHandler class. This class must be extended to implement the protocol-specific functionality. The URLConnection class represents a connection to an object referenced by a URL. The implementation of this class also is protocol-specific. It is this class that determines what type of content is in the file referenced by the URL.

For each type of file format, (such as GIF or HTML) there is an associated MIME-type/subtype combination. The ContentHandler class must be extended for each MIME-type. Figure 11-2 shows the basic relationships between URLs, ProtocolHandlers, and ContentHandlers.

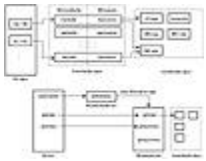

**Figure 11-2** URL class relationships

The basic functionality required of a ContentHandler is very small. A ContentHandler object should be able to read data off a URLConnection and construct an object that represents that content type. So if there is a subclass of Image that represents a GIF image, a ContentHandler for GIF files would simply read the data off the URLConnection and return a GIFImage object.

The URLStreamHandlerFactory and ContentHandlerFactory interfaces may be implemented so that the protocol-specific and content-specific classes can be constructed within the factory object. This provides a simple, uniform interface that the Java classes use to manufacture different protocol-specific or content-specific classes without explicitly specifying the class name. All these classes and interfaces provide a useful abstraction for developers of applications, such as Web browsers. ContentHandlers provide a means by which even nonstandard data formats can be viewed without having a viewer installed on your local machine. Using Java, you get the viewer bundled along with the data!

Implementing a protocol handler or a content handler is highly specific to the nature of the protocol or the format of the data and is beyond the scope of this book. As a consequence, some of the method descriptions are not accompanied by concrete examples. The HotJava browser uses implementations of protocol and content handlers

written in Java. If you develop a protocol handler or a content handler, you will need to refer to the browser's documentation to determine the naming policy for these classes and also to determine where these classes must be installed so that the browser uses your protocol and content handlers.

## URL and Networking Exception Summaries

Exceptions signal abnormal error conditions within the application. In object-oriented terminology, a method that detects an error may throw (or generate) an exception. In order to detect this error condition, the application that invoked the method must catch the exception. The Java keywords *try* and *catch* are used to detect exceptions. Exceptions are usually caught to inform the user that something bad happened. Exceptions related to URLs and networking are described at the end of the summary section.

Table 11-1 summarizes the classes and interfaces described in this chapter.

**Table 11-1** Class/interface descriptions

| Class/Interface | Description |
|---|---|
| ContentHandler | Interprets data read from a URLConnection object, and constructs an object that represents a specific MIME-type/subtype combination, such as image/gif, text/plain, etc. |
| ContentHandlerFactory | Defines the interface that must be implemented by classes that know how to create an instance of the subclass of ContentHandler that handles the specified MIME-type. |
| URL | Represents a Uniform Resource Locator (URL). Uniform Resource Locators are references to objects on the World Wide Web that can be retrieved by using protocols such as the HyperText Transfer Protocol (HTTP). |
| URLConnection | Sets or modifies the connection-session parameters and handles the network connection to the remote object that is referred to by a Uniform Resource Locator. |
| URLEncoder | Encodes strings into URL format. Encoding the strings in this uniform format ensures that the string is not corrupted by errors, such as character set variations on different systems (when transmitted over the network). |
| URLStreamHandler | Specifies an abstract base class that must be |

| | |
|---|---|
| | subclassed to implement stream handlers for specific protocols, such as http, nntp, ftp, etc. |
| URLStreamHandlerFactory | Defines the interface that must be implemented by a class that knows how to create an instance of a specific subclass of URLStreamHandler that handles a protocol. |
| MalformedURLException | Signals that the specified Uniform Resource Locator (URL) is invalid. |
| ProtocolException | Indicates that an EPROTO error was detected when the application tried to connect to a socket. |
| SocketException | Indicates that an error occurred while performing an operation on a socket. |
| UnknownHostException | Indicates that the address of the host specified by a network client is not valid. |
| UnknownServiceException | Signals an error indicating that the requested service is not supported by the client-server protocol. |

## ContentHandler

**Purpose**

To interpret data read from a URLConnection object and construct an object that represents a specific MIME-type/subtype combination, such as image/gif, text/plain, and so on.

**Syntax**

public class ContentHandler extends Object

**Description**

This abstract class must be subclassed to implement content handlers for specific MIME-type/subtype combinations. ContentHandlers read data from the URLConnection stream and construct an object that represents the MIME-type. Subclasses of this class can be used in Web browsers to interpret specific MIME-type/subtype combinations. Figure 11-2 shows the basic relationships between URLs, ProtocolHandlers and ContentHandlers. Applications should not construct ContentHandlers directly. Instead, they should use the getContent methods of the URL class or the URLConnection class. This method constructs and returns an instance of a ContentHandler object that is appropriate for the MIME-type of the connection. By default, the URLConnection class combines the MIME-type and subtype to form a path name for the Java class. It then looks for a Java .class file of this name in the sun/net/www/content directory (if such a directory is found relative to the directory in which the standard Java classes were installed). For example, if the MIME-type/sub-type returned were image/gif, the default content

handler for this object would be sun/net/www/content/image/gif.class. Figure 11-3 shows the inheritance diagram for the ContentHandler class.


**Figure 11-3** Inheritance diagram for the ContentHandler class

**PackageName**

*java.net*

**Imports**

*import java.net.ContentHandler;*

**Constructors**

public ContentHandler()

**Parameters**

None.

**Example**

The following example shows how you can write your own content handlers.

```
// File: tdif.java
//       A Java class that represents an image that conforms to a
         fictitious image format known
//       as Three D Image Format. Files of this type will have the
         .tdi extension
//       [Note: This class is not complete]
package CustomPackage;
import java.awt.Image;
import java.net.*;
import java.io.*;
public class tdif extends Image {
         // Different forms of constructors for this object
         ....
         // This constructor constructs a tdif object by reading
         data from a URLConnection stream
         public tdif(URLConnection uc) {
                  try {
                           DataInputStream data = new
                           DataInputStream(uc.getInputStream());
                           // read the data and construct the tdif
                           object
                           ....
                           data.close();
                  } catch (IOException e){
                           System.out.println("Caught
                           exception!");
                  }
         }
         // Implementation of the various methods of the Image
         class are added here
         .....
```

```
}

// File: tdi.java    ContentHandler class for image/tdi objects
//       For the purposes of this example, assume that the
         MIME-type/subtype combination for objects of
//       Three D Image Format are image/tdi.
//       Thus, any objects of this format will be handled by the
         sun/net/www/content/image/tdi.class
//       content handler class. This class just creates and returns
         a tdif object.

package sun.net.www.content.image;
import CustomPackage.tdif;

public class tdi extends ContentHandler {
         public Object getContent(URLConnection uc) throws
         IOException {
                  // if there is an error, then throw an IOException
                  // otherwise create and return the appropriate object
                  return new tdif(uc);
         }
}
```

## getContent(URLConnection)

**ClassName**

ContentHandler

**Purpose**

Reads and interprets the data from a URLConnection stream and constructs an object that represents the MIME-type/subtype combination handled by this class.

**Syntax**

public abstract Object getContent(URLConnection urlc) throws IOException

**Parameters**

***urlc***

The URLConnection stream from which data must be read and interpreted to create an object for the MIME-type that this class represents.

**Description**

This abstract method must be implemented by each subclass of ContentHandler to read and interpret data from the URLConnection data-stream and create a representation of the MIME object that the subclass represents. An IOException is thrown if some error occurs while reading the data from the URLConnection stream.

**Imports**

*import java.net.ContentHandler;*

**Returns**

This method returns an Object that represents the specific MIME-type/subtype implemented by this class.

**See Also**

The getContent method of the URL class and the getContent of the URLConnection class described in this chapter

**Example**

The previous example illustrates how you can implement this method in your own content handlers.

## ContentHandlerFactory

**Purpose**

Defines the interface that must be implemented by classes that know how to create an instance of a subclass of ContentHandler that handles a specified MIME-type.

**Syntax**

public interface ContentHandlerFactory extends Object

**Description**

As the name implies, a class that implements this interface must know how to manufacture (construct) instances of subclasses of the ContentHandler class. Each concrete subclass of the abstract base class, ContentHandler, handles a specific MIME-type. A class that implements this interface must maintain an association between the MIME-type strings and the name of the class that handles that MIME-type/subtype combination. ContentHandlerFactory objects construct (on demand) the appropriate ContentHandler for a given MIME-type. This class will mainly be used in subclasses of the URLConnection class. If you are writing content handlers and you do not want to install these classes in the default directories, such as sun/net/www/content/, then you can use a class that implements the ContentHandlerFactory interface to create instances of your classes. To implement a protocol handler in Java, you must extend the URLStreamHandler class to provide implementations of streams that are specific to the protocol. You must also subclass the URLConnection class to provide implementations of the different types of content that can be handled by the protocol. The URLConnection class has a static ContentHandlerFactory object as a member variable. This implies that every instance of the URLConnection class uses the same source for constructing ContentHandlers. Figure 11-4 shows the inheritance diagram for the ContentHandlerFactory interface.

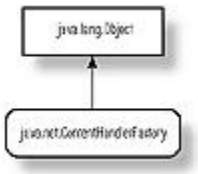


**Figure 11-4** Inheritance diagram for the ContentHandlerFactory interface

**PackageName**

*java.net*

**Imports**

*import java.net.ContentHandlerFactory;*

**Constructors**

None.

**Parameters**

None.

**Example**

The example for the createContentHandler method illustrates how you can implement a custom Content Handler factory class.

## createContentHandler(String)

**InterfaceName**

ContentHandlerFactory

**Purpose**

Constructs an instance of the specific subclass of ContentHandler that handles the specified MIME-type.

**Syntax**

public abstract ContentHandler createContentHandler(String mimetype)

**Parameters**

***mimetype***

The Multipurpose Internet Mail Extension (MIME) type for which an instance of the appropriate subclass of ContentHandler is to be constructed.

**Description**

Classes that implement the ContentHandlerFactory interface must know the names and locations of the specific subclasses of ContentHandler that handle the various MIME-type/subtype combinations. Depending on the MIME-type/subtype specified, an instance of the appropriate subclass of ContentHandler is constructed and returned. How does the URLConnection class know which MIME-type is to be supplied as a parameter to this method? The getContentType method of the URLConnection class provides the answer to this question. The getContentType method returns a string that contains the MIME-type/subtype combination of the URLConnection. This string can then be used as the parameter to this method.

**Imports**

*import java.net.ConentHandlerFactory;*

**Returns**

This method returns an instance of the ContentHandler subclass that represents the specified MIME-type.

**See Also**

The ContentHandler class; the setContentHandlerFactory method of the URLConnection class; and the getContentType method of the URLConnection class, all of which are described in this chapter

**Example**

The following example shows you a class that implements this interface.

```
// File: CustomContentHandlerFactory.java
//       A Java class that implements the ContentHandlerFactory
interface
//       This factory looks for classes in the CustomPackage/content
         directory
```

```
//       It constructs the class name by prepending this "root"
directory
         to the
//       MIME-type/sub-type of the object

package CustomPackage;
import java.net.*;

public class CustomContentHandlerFactory implements
ContentHandlerFactory
{
        public ContentHandler createContentHandler(String mimeType) {
                String rootDir = "CustomPackage.content.";
                // The mimeType string is of the form image/gif
                // Replace the slash (/) with a period (.)
                String mimeClass = mimeType.replace('/', '.');
                String fullClassname = rootDir + mimeClass;
                // create and return the ContentHandler object
                return (ContentHandler) Class.forName(fullClassname).
                newInstance();
        }
}
```

## URL

**Purpose**

Represents a Uniform Resource Locator (URL). Uniform Resource Locators are references to objects on the World Wide Web that can be retrieved using protocols such as the HyperText Transfer Protocol (HTTP).

**Syntax**

public final class URL extends Object

**Description**

This class is used to access objects on the World Wide Web. It encapsulates the concept of a Uniform Resource Locator. Handling the different protocol types (http, ftp) and content types (GIF images, Postscript files) is transparent to the application that uses URL objects to access data on the Web. This class cannot be subclassed, and once constructed, the URL object cannot be modified (i.e., instances of the URL class are constant objects). You can create URL objects either by specifying the absolute path or by specifying a path relative to another URL. Constructing URL objects by specifying a path relative to another URL object is useful for creating URLs to references (named anchor tags, e.g., "http://www.syr.edu/index.html#LIBRARY") within a HTML (HyperText Markup Language) file. If the parameters supplied to the constructor are not valid, a MalformedURLException is generated, and hence the constructor statements should be within a try/catch statement pair. The URL class maintains a table of URLStreamHandler objects that handle different protocols such as http, file, news, doc, and verbatim. These URLStreamHandler objects are created on demand and are shared by all instances of the URL class (i.e., the table is a static

member variable of the URL class). When a URL object is created, a URLStreamHandler object is created. This specific instance of the URLStreamHandler object depends on the protocol specified in the URL. Figure 11-2 shows the basic relationships between URLs, ProtocolHandlers, and ContentHandlers. If a URLStreamHandler factory object has been defined, then this factory is used to generate the URLStreamHandler. If no factory has been defined, then the URL class looks in certain default directories (for example: sun/net/www/protocol/http/Handler.class for the http protocol) for the URLStreamHandler subclass that implements the specified protocol. If you are writing Java applications that access resources on the World Wide Web, then you will be using the URL class often. Figure 11-5 shows the inheritance diagram for the URL class.

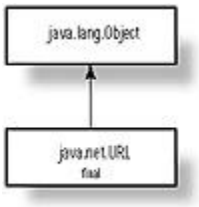


**Figure 11-5** Inheritance diagram for the URL class

**PackageName**

*java.net*

**Imports**

*import java.net.URL;*

**Constructors**

public URL(String *protocol*, String *host*, int *port*, String *file*) throws MalformedURLException
public URL(String *protocol*, String *host*, String *file*) throws MalformedURLException
public URL(String *spec*) throws MalformedURLException
public URL(URL *context*, String *spec*) throws MalformedURLException

**Parameters**

***protocol***

The protocol (http, news, and so on) to use for this URL.

***host***

The Internet name of the host machine to connect to (e.g., www.syr.edu).

***port***

The port number on the *host* machine to connect to.

***file***

The path name of the file on the host.

***spec***

A string specifying an absolute (unparsed) URL (e.g. "http://www.syr.edu/").

***context***

A URL object to be used as a context into which a string specifying a URL (usually a relative path to the context URL) may be parsed.

**Example**

The following samples of code illustrate the different ways of constructing URL objects. Refer to the URLTestControls constructor method in Step 7 of the project at the end of this chapter for a more complete example.

```
....
        try {
                URL u1 = new URL("http://www.syr.edu/");   // absolute
                URL
                URL u2 = new URL(u1, "index.html");        // relative
                URL
                specification
                URL u3 = new URL("http", "www.syr.edu", 80,
"/index.html"
                );
        } catch (MalformedURLException e) {
        ...                                                // error
                                                           processing
}
```

## equals(Object)

**ClassName**

URL

**Purpose**

Compares the specified URL object to the URL on which this method is invoked and determines whether the two URL objects are equal or not.

**Syntax**

public boolean equals(Object obj)

**Parameters**

***obj***

The URL object to compare against.

**Description**

The URL object specified in the parameter of this method is compared to the URL object on which the method was invoked. The protocol, hostname, port number, and file specifications of two URL objects are compared.

Note: This method does not make use of facilities like the Unix Domain Name Service (DNS) to look for aliases of the hostnames. It simply compares the strings specifying the hostname, protocol, and file specifications. If the hostname strings do not match, this method evaluates the Internet addresses of the hostnames of the URL objects being compared.

**Imports**

*import java.net.URL;*

**Returns**

This method returns true if and only if the parameter *obj* is an instance of the URL class and the protocol, host, port, and file properties of the two URL objects match.

**Example**

The following Java class illustrates the use of this method.

```
import java.net.*;
```

```
public class URLEqualsTest {
    public static void main(String args[]) {
        try {
            URL u1 = new URL("http","web.syr.edu", 80, "/index.html");
            URL u2 = new URL("http://www.syr.edu");
            URL u3 = new URL("http://web.syr.edu:80/
            index.html");
            if (u1.equals(u2))
                System.out.println("URL u1 = URL u2");   // should NOT
                print this
            if (u1.equals(u3))
                System.out.println("URL u1 = URL u3");   // should
print
                this
            if (u2.sameFile(u3))
                System.out.println("URL u2 = URL u3");   // should NOT
                print this
        } catch (MalformedURLException e) {
            System.out.println("Caught exception !");
        }
    }
}
```

## getContent()

**ClassName**

URL

**Purpose**

Returns an object constructed from the data read from the object referred to by this URL.

**Syntax**

public final Object getContent() throws IOException

**Parameters**

None.

**Description**

The protocol-specific URLStreamHandler handling this URL determines the content type of the object referred to by this URL. The ContentHandler for that MIME-type reads the data from the URLConnection stream and constructs the content-specific object that is returned.

**Imports**

*import java.net.URL;*

**Returns**

The Object returned by this method is constructed from the data read from the URLConnection. Depending on the MIME-type of the object referred to by this URL, a specific instance of a class that represents that content is created. The instance of operator should be used to determine the class that this object belongs to. An IOException indicates that an error occurred while reading and interpreting data from the remote object.

**See Also**

The ContentHandler class; the getContent method; and the URLConnection class, described in this chapter

**Example**

The following example shows how you can use the instance of operator to determine the type of object that the URL refers to.

```
// File: URLContentType.java

import java.net.*;
import java.io.IOException;

public class URLContentType {
    public static void main(String args[]) {
                try {
                        // Replace the following URL with a URL
                        to any text file
                        URL u = new URL("http://cosmos/
                        ~asriniva/hello.txt");
                        try {
                                Object o = u.getContent();
                                // Check the type of object
                                if (o instanceof String) {
                                        // Use this object as a String
                                        String s = (String) o;
                                        System.out.println( "This URL
                                        refers to a String object");
                                        System.out.println ("Contents
of
                                        the URL object: " + s);

                                }

                        } catch (IOException ie) {
                                     System.out.println ("Caught IOE
!"
                                     );

                        }
        } catch (MalformedURLException e) {
            System.out.println("Caught MUE !");
        }
    }
}
```

This program prints the following on the screen:

```
This URL refers to a String object
Contents of the URL object: This is the only line of text in hello.txt!
```

## getFile()

**ClassName**

URL

**Purpose**

Gets the filename portion of the URL specification.

**Syntax**

public String getFile()

**Parameters**

None.

**Description**

This method returns the value of the filename portion of the URL specification. The first two forms of the URL constructor allow you to specify the filename explicitly. The next two variants of the URL constructor allow you to specify a URL as a string of text (e.g., "http://www.syr.edu/index.html") and the URL constructor parses this string to extract the filename ("/index.html" in this case).

**Imports**

*import java.net.URL;*

**Returns**

The return value of this method is a String object that contains the name of the file referenced by this URL object.

**See Also**

Constructors for this class

**Example**

Refer to the showParams() method in the URLPanel class of the section project (Step 5) at the end of this chapter.

## getHost()

**ClassName**

URL

**Purpose**

Retrieves the hostname portion of the Uniform Resource Locator specification.

**Syntax**

public String getHost()

**Parameters**

None.

**Description**

This method returns the value of the hostname portion of the URL specification. The first two forms of the URL constructor allow you to specify the hostname explicitly. The next two variants of the URL constructor allow you to specify a URL as a string of text (e.g., "http://www.syr.edu/index.html") and the URL constructor parses this string to extract the hostname ("www.syr.edu" in this case).

**Imports**

*import java.net.URL;*

**Returns**

The return value of this method is a String object that contains the hostname on which the file referenced by this URL object resides.

**See Also**

Constructors for this class

**Example**

This use of this method is illustrated in the showParams() method in the URLPanel class of the section project (Step 5) at the end of this chapter.

## getPort()

**ClassName**

URL

**Purpose**

Gets the port number (on the target host machine) to which this URL object connects.

**Syntax**

public int getPort()

**Parameters**

None.

**Description**

This method returns the value of the port number of the URL specification. The first form of the URL constructor allows you to explicitly specify the port number. The second form of the URL constructor sets the port number to the default value (which depends on the protocol). The next two variants of the URL constructor allow you to specify a URL as a string of text (e.g., "http://www.syr.edu:80/index.html") in which you can optionally specify a port number, and the URL constructor parses this string to extract the port number (80 in this case).

**Imports**

*import java.net.URL;*

**Returns**

This method returns an integer that specifies the port number this URL object connects to. If the port number was not explicitly specified while constructing the URL, then the return value of this method is -1.

**See Also**

Constructors for this class

**Example**

The use of this method is illustrated in the showParams() method in the URLPanel class of the section project (Step 5) at the end of this chapter.

## getProtocol()

**ClassName**

URL

**Purpose**

Gets the protocol used to retrieve the object referenced by this URL.

**Syntax**

public String getProtocol()

**Parameters**

None.

**Description**

This method returns the value of the protocol portion of the URL specification. The first two forms of the URL constructor allow you to specify the protocol explicitly. The next two variants of the URL constructor allow you to specify a URL as a string of text (e.g., "http://www.syr.edu/index.html") and the URL constructor parses this string to extract the protocol ("http" in this case).

**Imports**

*import java.net.URL;*

**Returns**

The return value of this method is a String object that specifies the protocol used to retrieve the object referenced by this URL.

**See Also**

Constructors for this class

**Example**

The use of this method is illustrated in the showParams() method in the URLPanel class of the section project (Step 5) at the end of this chapter.

## getRef()

**ClassName**

URL

**Purpose**

Gets the anchor tagname for this URL object. Anchors are used to point to a specific section in a document.

**Syntax**

public String getRef()

**Parameters**

None

**Description**

The # mark indicates a named anchor in a HTML (HyperText Markup Language) document. By specifying an anchor in a document, one can go directly to that specified section in the document. This method retrieves the name of the anchor (if any) from the URL specification.

Note: This method does not detect the reference specification when the URL object is created using the following form of the URL constructor.

URL u1 = new URL("http://www.syr.edu/index.html#LIBRARY");

In this case, u1.getRef() returns the null string.

**Imports**

*import java.net.URL;*

**Returns**

This method returns the anchor tagname (string following the '#' character) in the URL.

**See Also**

The set method of this class and the different forms of constructors for this class

**Example**

The use of this method is illustrated in the showParams() method in the URLPanel class of the section project (Step 5) at the end of this chapter.

## hashCode()

**ClassName**

URL

**Purpose**

Returns a hash value that can be used to index into a hash table.

**Syntax**

public int hashCode()

**Parameters**

None.

**Description**

This method overrides the hashCode method of the Object class. It returns a number that is the hash value of the URL object on which this method was invoked.

**Imports**

*import java.net.URL;*

**Returns**

This method returns an integer that represents the hash value for this URL object.

**Example**

The following Java class shows how to invoke this method in your application.

```
import java.net.*;

public class URLHashTest {
    public static void main(String args[]) {
        try {
            URL u1 = new URL("http://web.syr.edu/");
            System.out.println("u1.hashCode = " + u1.hashCode());
        } catch (MalformedURLException e) {
            System.out.println("Caught exception !");
        }
    }
}
```

## openConnection()

**ClassName**

URL

**Purpose**

Opens a connection to the object referred to by this URL object.

**Syntax**

public URLConnection openConnection() throws IOException

**Parameters**

None.

**Description**

The subclass of URLStreamHandler that handles the protocol specified by this URL object creates an instance of a subclass of the URLConnection class and returns this URLConnection object. The specific subclass of the URLConnection class to instantiate is determined by the URLStreamHandler object that implements the protocol used by this URL object. An IOException is thrown when there is an error in establishing a connection to the remote object.

**Imports**

*import java.net.URL;*

**Returns**

This method returns an instance of the protocol-specific subclass of the URLConnection class that contains a connection to the object referred to by this URL.

**See Also**

The URLConnection class; the openStream and getContent methods of the URL class, described in this chapter

**Example**

This example illustrates how you can access the URLConnection object associated with a URL and read data from the URLConnection object over the network.

```
// File: URLRead1.java
//       A class that reads data from a URL using a URLConnection
object

import java.net.*;
import java.io.*;

public class URLRead1 {
    public static void main(String args[]) {
                try {

                        // You can replace this URL with one
                        pointing to any
                        // HTML file
                        URL u = new URL("http://cosmos/
                        ~asriniva/test.html");
                        try {
                             // get a handle to the URLConnection
object
                             // for this URL
                             URLConnection uc =  u.openConnection();

                             // Check whether this URL object
                             can be "read"
                             if (uc.getDoInput() == true) {

                                      // Attach an input stream
                                      and read from this
                                      // input stream
                                      InputStream iStream =
                                      uc.getInputStream();
                                      DataInputStream data =
```

```
                                        new
                                        DataInputStream(iStream);
                                        String line;
                                        int lineNumber = 1;
                                        // Print each line in the
                                        file, prefixing it
                                        // with the corresponding
                                        line number
                                        while ((line =
                                        data.readLine())
                                        != null) {
                                             System.out.println
                                               (“Line”
                                               + lineNumber
                                               + “: ” +
                                               line);
                                               lineNumber++;
                                        }
                                        data.close();   // close
                                        the input stream
                                   }

                              } catch (IOException ie) {
                                  System.out.println(“Caught IOE !”);
                              }
                    } catch (MalformedURLException e) {
                        System.out.println(“Caught MUE !”);
                    }
          }
}
```

The file test.html contains:

```
<HTML>
<HEAD>
<TITLE> A Sample HTML file </TITLE>
</HEAD>
<BODY>
<P> This is the first and only paragraph in this file. This file
contains 10 lines.
</BODY>
</HTML>
```

This example prints the following lines on the screen:

```
Line 1: <HTML>
Line 2: <HEAD>
Line 3: <TITLE> A Sample HTML file </TITLE>
Line 4: </HEAD>
Line 5: <BODY>
Line 6: <P>
Line 7: This is the first and only paragraph in this file. This file
Line 8: contains 10 lines.
Line 9: </BODY>
Line 10: </HTML>
```

## openStream()

### ClassName

URL

**Purpose**

Opens an input stream to the object referenced by this URL.

**Syntax**

public final InputStream openStream() throws IOException

**Parameters**

None.

**Description**

This function returns an InputStream to the object referred to by the URL. The InputStream is established by the protocol-specific URLConnection object that was created by the URLStreamHandler object handling this URL. If the protocol of this URL object supports input streams, then applications can use this InputStream object to read the data of the object referred to by the URL.

**Imports**

*import java.net.URL;*

**Returns**

The InputStream object returned by this method can be used to read the data of the object referred to by this URL object. If the protocol does not support InputStreams, an UnknownServiceException is thrown. Protocol implementors can trigger other exceptions while implementing the getInputStream method of the protocol-specific subclass of the URLConnection class.

**See Also**

The getInputStream method of the URLConnection class; the openConnection and getContent methods of the URL class described in this chapter

**Example**

This example essentially performs the same function as the previous example. It uses the openStream method of the URL class to read data from the object. This example uses the same test.html file as the previous example.

```
// File: URLRead2.java
//       A class that reads data from a URL using the URL's
openStream method

import java.net.*;
import java.io.*;

public class URLRead2 {
    public static void main(String args[]) {
                try {
                        // You can replace this URL with one
                        pointing to any
                        // HTML file
                        URL u = new URL("http://cosmos/
                        ~asriniva/test.html");
                        try {
                                // Attach an input stream and read
                                from this
                                // input stream
                                DataInputStream data = new
                                DataInputStream(u.openStream());
                                String line;
                                int lineNumber = 1;
```

```
                                    // Print each line in the file,
                                    prefixing it
                                    // with the corresponding line
                                    number
                                    while ((line = data.readLine()) !=
                                    null) {
                                            System.out.println("Line "
                                           + lineNumber + ": " + line);
                                           lineNumber++;
                                    }
                                    data.close();  // close the input
                                    stream

                            } catch (IOException ie) {
                                         System.out.println("Caught
                                         IOE !");
                            }
                } catch (MalformedURLException e) {
                    System.out.println("Caught MUE !");
                }
            }
}
```

When executed, this example prints the following lines on the screen:

```
Line 1: <HTML>
Line 2: <HEAD>
Line 3: <TITLE> A Sample HTML file </TITLE>
Line 4: </HEAD>
Line 5: <BODY>
Line 6: <P>
Line 7: This is the first and only paragraph in this file. This file
Line 8: contains 10 lines.
Line 9: </BODY>
Line 10: </HTML>
```

## sameFile()

**ClassName**

URL

**Purpose**

Compares the specified URL object against the object on which this method was invoked.

**Syntax**

public boolean sameFile(URL other)

**Parameters**

***other***

The URL object that must be compared against the object on which this method is invoked.

**Description**

If the four fields ( protocol, hostname, port number, and file path name) of the two URL objects are the same, then the URLs are said to be equal. The reference field, that indicates an offset in a file, is not taken into account for the comparison.

**Imports**

*java.net.URL;*

**Returns**

This method returns true if the specified URL object is equal to this URL; otherwise, it returns the boolean value false.

**See Also**

The equals and set methods of the URL class, described in this chapter

**Example**

This method is used in the example for the equals method of this class.

## set(String, String, int, String, String)

**ClassName**

URL

**Purpose**

Sets the individual fields of the URL object. This is a privileged function that is accessible only to classes in the *java.net* package.

**Syntax**

protected void set(String *protocol*, String *host*, int *port*, String *file*, String *ref*)

**Parameters**

***protocol***

The protocol (http, news, etc) to use for this URL.

***host***

The Internet name of the host machine to connect to (e.g., www.syr.edu).

***port***

The port number on the *host* machine to connect to.

***file***

The path name of the file on the host.

***ref***

The name of the reference that indicates a specific offset into the *file.*

**Description**

This method is provided so that the parseURL method of the URLStreamHandler class can set the fields of a URL. It is a protected method that is accessible only to classes in the *java.net* package.

**Imports**

*import java.net.URL;*

**Returns**

None.

**Example**

This method can only be invoked by classes in the *java.net* package and hence no example code illustrating the usage of this method is provided here.

## setURLStreamHandlerFactory(URLStreamHandlerFactory)

**ClassName**

URL

**Purpose**

Specifies the factory object (that implements the URLStreamHandlerFactory interface) that all URL objects should use to create protocol-specific URLStreamHandler objects.

**Syntax**

public static synchronized void setURLStreamHandlerFactory(URLStreamHandlerFactory fac)

**Parameters**

***fac***

The URLStreamHandlerFactory object that must be used for creating protocol-specific stream handlers.

**Description**

All objects of the URL class share the same URLStreamHandlerFactory object. By invoking this method, you can install your own URLStreamHandlerFactory. An error is generated if a URLStreamHandlerFactory already exists. If you write new protocol handler classes and install these classes in nonstandard locations, you will need to create a class that implements the URLStreamHandlerFactory interface. This factory class will need to know where to find the protocol-specific implementations of classes such as the URLConnection class and the URLStreamHandler class.

**Imports**

*import java.net.URL;*

**See Also**

The URLStreamHandlerFactory class and the URLStreamHandler class described in this chapter

**Returns**

None.

**Example**

The following pieces of code illustrate how you might set the URLStreamHandlerFactory to a custom factory that you have implemented.

```
// File: myFactory.java
import java.net.*;

public class myFactory implements URLStreamHandlerFactory {
         // implement URLStreamHandlerFactory interface methods
         here
         .....
         public URLStreamHandler createURLStreamHandler(String
         protocol) {
                 .....
                 // return the appropriate stream handler for the
                 protocol
         }
         .....
}

// File: URLSetFactoryTest.java
import java.net.*;
```

```
public class URLSetFactoryTest {
        public static void main(String args[]) {
                   try {
                URL u1 = new URL("http://web.syr.edu/");
                u1.setURLStreamHandlerFactory(new MyFactory());

                   } catch (MalformedURLException e) {
                        System.out.println("Caught exception !");
                }
        }
}
```

## toExternalForm()

**ClassName**

URL

**Purpose**

Represents this URL as a text string.

**Syntax**

public String toExternalForm()

**Parameters**

None.

**Description**

The textual representation of this URL is constructed from the individual fields of the URL (such as protocol, hostname, etc.). Default values (such as default port number) are omitted from the text string.

**Imports**

*import java.net.URL;*

**Returns**

The String object returned by this method contains a text string that specifies the protocol, hostname, portnumber (if specified), file name, and reference (if specified) of this URL object.

**See Also**

The toString method of this class

**Example**

This method is invoked by the toString method of this class. The following Java class shows how to invoke this method in your application.

```
import java.net.*;

public class URLTestExternalForm {
    public static void main(String args[]) {
        try {
            URL u1 = new URL("http", "web.syr.edu", 80, "/index.html");
            System.out.println("u1.toExternalForm = " + u1.toExternal
            Form());
        } catch (MalformedURLException e) {
            System.out.println("Caught exception !");
        }
    }
}
```

When this example is compiled (javac URLTestExternalForm.java) and executed (java URLTestExternalForm), the following string is printed on the screen.

```
u1.toExternalForm = http://web.syr.edu:80/index.html
```

## toString()

**ClassName**

URL

**Purpose**

Represents this URL as a text string.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method simply invokes the toExternalForm method to represent the URL object as a text string.

**Imports**

*java.net.URL;*

**Returns**

The String object returned by this method contains a text string that specifies the protocol, hostname, port number (if specified), file name, and reference (if specified) of this URL object.

**See Also**

The toExternalForm method of this class

**Example**

The use of this method is illustrated in the showParams method in the URLPanel class of the section project (Step 5) at the end of this chapter.

## URLConnection

**Purpose**

Sets/modifies the connection-session parameters and handles the network connection to the remote object referred to by a Uniform Resource Locator.

**Syntax**

public class URLConnection extends Object

**Description**

This abstract class must be subclassed by protocol-implementors to provide protocol-specific implementations for connecting to remote objects referred to by Uniform Resource Locators (URLs). This class will handle the parsing of protocol-specific message headers and message content. The various properties of the connection session for a protocol will also be handled by subclasses of this class. All instances of the URLConnection class share a table of ContentHandlers. These ContentHandlers represent specific MIME-type/subtype combinations. Instances of this class (for a specific protocol) are created by the subclass of the URLStreamHandler class that implements the specified protocol's stream

handling functionality. Figure 11-2 shows the basic relationships between URLs, ProtocolHandlers, and ContentHandlers. Figure 11-6 shows the inheritance diagram for the URLConnection class.

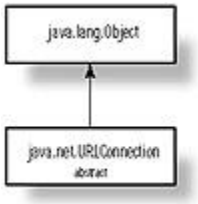


**Figure 11-6** Inheritance diagram for the URLConnection class

**PackageName**

*java.net*

**Imports**

*import java.net.URLConnection;*

**Constructors**

protected URLConnection(URL *url*)

**Parameters**

***url***

The URL object to which a connection needs to be established.

**Example**

The constructor method for this class is protected and hence you cannot create objects of this class. To implement protocol handlers, you will need to extend this class.

## connect()

**ClassName**

URLConnection

**Purpose**

Connects to the remote object referred to by the URL for which this URLConnection object was created.

**Syntax**

public abstract void connect() throws IOException

**Parameters**

None.

**Description**

Invoking this method causes a connection to be established to the object referred to by the URL. The properties of this connection-session cannot be altered (using the methods such as setDoInput and setDoOutput) once a connection is established. This method must be implemented by any class that extends the URLConnection class.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**Example**

The following code shows a portion of a Java class that extends the URLConnection class.

```
// File: CustomURLConnection.java
//        URLConnection object for a new protocol
package CustomProtocolConnection;
import java.net.*;

public class CustomURLConnection extends URLConnection {
        // code for the URLConnection class methods that implement
        the specifics of the
        // protocol is written here
        ....
        public void connect() throws IOException {
               ....
               // Implement the policies of the protocol to connect
               to the
               // object specified by the URL
               ....
        }
        ...
}
```

## getAllowUserInteraction()

**ClassName**

URLConnection

**Purpose**

Returns the value of the flag that indicates whether the protocol permits user interaction while establishing the connection to the remote object.

**Syntax**

public boolean getAllowUserInteraction()

**Parameters**

None.

**Description**

Protocols such as http, that have access/security control features, allow user interaction (such as authentication by asking for a user name and password) during the process of setting up a connection to an object referred to in a URL. Protocol implementors specify whether this interaction is permitted by the protocol or not.

**Imports**

*import java.net.URLConnection;*

**Returns**

The return value is true if the protocol permits user-interaction at the time of establishing a connection, and false if user-interaction is not permitted by the protocol.

**See Also**

The setAllowUserInteraction, getDefaultAllowUserInteraction and setDefaultAllowUserInteraction methods of this class

**Example**

The following example illustrates the usage of this method and some of the other related methods of this class.

```
// File: URLConnTest1.java
//       Illustrates the usage of some of the methods of the
URLConnection class

import java.net.*;
import java.io.IOException;

public class URLConnTest1 {
    public static void main(String args[]) {
        try {
       // Replace the following URL with any valid URL
       URL u1 = new URL("http://cosmos/~asriniva/index.html");
            try {
                URLConnection uc = u1.openConnection();
                System.out.println("URL: " + u1.toString());
                System.out.println("getAllowUserInteraction: " +
                                   uc.getAllowUserInteraction());
                System.out.println("getDefaultAllowUserInteraction: " +

uc.getDefaultAllowUserInteraction());
                System.out.println("getUseCaches: " +
                                   uc.getUseCaches());
                System.out.println("getDefaultUseCaches: " +
                                   uc.getDefaultUseCaches());
                System.out.println("getDoInput: " +
                                   uc.getDoInput());
                System.out.println("getDoOutput: " +
                                   uc.getDoOutput());
     } catch (IOException ie) {
                System.out.println("Caught IOException !");
            }
        } catch (MalformedURLException e) {
            System.out.println("Caught MUE !");
        }
    }
}
```

When this program is compiled and executed it prints the following messages on the screen.

```
URL: http://cosmos/~asriniva/index.html
getAllowUserInteraction: false
getDefaultAllowUserInteraction: false
getUseCaches: true
getDefaultUseCaches: true
getDoInput: true
getDoOutput: false
```

## getContent()

**ClassName**

URLConnection

**Purpose**

Reads the data from the remote object and constructs an instance of the subclass of ContentHandler that handles the MIME-type of the object this URL refers to.

**Syntax**

public Object getContent() throws IOException

**Parameters**

None.

**Description**

This method determines the MIME-type/subtype of the object referred to by the URL and constructs the appropriate ContentHandler (if it does not already exist). The getContent method of the ContentHandler class is invoked and the ContentHandler object reads the data from this URLConnection stream. Then it constructs and returns the object referred to by the URL. For example, a plain text content handler may just read the data from the URLConnection object and return a String object containing the text read from the remote object, or a content handler that handles the image/gif MIME-type could construct a GIFImage object. This GIFImage object would typically be an instance of a subclass of the Image class.

**Imports**

*import java.net.URLConnection;*

**Returns**

The Object returned by this method is constructed from the data read from the URLConnection. Depending on the MIME-type of the object referred to by the URL, a specific instance of a class that represents that content is created. The instanceof operator should be used to determine the class that this object belongs to. An IOException indicates that an error occurred while reading and interpreting data from the remote object.

**See Also**

The ContentHandler class and the getContent method of the URL class and of the URLStreamHandler class. These are described in this chapter.

**Example**

The following example uses the getContent method of the URLConnection class to access the object referred to by the URL.

```
// File: URLConnContentType.java
import java.net.*;
import java.io.IOException;

public class URLConnContentType {
    public static void main(String args[]) {
                try {

                        // Replace the following URL with a URL
                        to any text file
                        URL u = new URL("http://cosmos/
                        ~asriniva/hello.txt");
                        try {
                                // get a handle to the
```

```
                                        URLConnection object
                                        // for this URL URLConnection
                                        uc = u.openConnection();

                                        Object o = uc.getContent();
                                        // Check the type of object
                                        if (o instanceof String) {
                                                  // Use this object as
                                                  a String
                                                  String s = (String) o;
                                                  System.out.println
                                                  (“This URL refers
                                                  to a String object”);
                                                  System.out.println
                                                  (“Contents of the URL
                                                  object: ” + s);

                                        }
                              } catch (IOException ie) {
                                        System.out.println(“Caught
                                        IOE !”);
                              }
                    } catch (MalformedURLException e) {
                        System.out.println(“Caught MUE !”);
                    }
          }
}
```

This program prints the following on the screen.

This URL refers to a String object

```
Contents of the URL object: This is the only line of text in hello.txt!
```

## getContentEncoding()

**ClassName**

URLConnection

**Purpose**

Gets the mechanism used to encode the data of the remote object to which this URLConnection is connected.

**Syntax**

public String getContentEncoding()

**Parameters**

None.

**Description**

One of the MIME header fields that specifies information about the object is the Content-Encoding field. It specifies the encoding mechanism used to encode the object data. Content codings are primarily used to allow data to be compressed or encrypted. Encoding mechanisms such as compress and gzip are used to compact the data so that less data needs to be transferred over the network.

**Imports**

*import java.net.URLConnection;*

**Returns**

This method returns a String that specifies the encoding mechanism used. If the encoding mechanism is not known, this method returns a null value. Two common encoding mechanisms are x-compress and x-gzip.

**Example**

The following example illustrates the usage of some of the methods of this class.

```
// File: URLConnTest.java
//       Illustrates the usage of some of the methods of the
URLConnection class
import java.net.*;
import java.io.IOException;
import java.util.Date;

public class URLConnTest {
    public static void main(String args[]) {
        try {
         // Replace the following URL with any valid URL
         URL u1 = new URL("http://cosmos/~asriniva/index.html");
          try {
             URLConnection uc = u1.openConnection();
             System.out.println("URL: " + u1.toString());
             System.out.println("Content-Type: " + uc.getContentType
             ());
             System.out.println("Content-Length: " +
                                uc.getContentLength());
             System.out.println("Content-Encoding: " +
                                uc.getContentEncoding());
          // Convert the date value to an ASCII string
      Date d1 = new Date(uc.getDate());
      System.out.println("Date (value): " +
                                  uc.getDate() + " (string): " +
                                  d1.toString());
          // Convert the last-modified date value to an ASCII string
      Date d2 = new Date(uc.getLastModified());
              System.out.println("Last modified on (value): " +
                                  uc.getLastModified() + " (string): "
+
                                  d2.toString());
          System.out.println("Expires on: " +
                                  uc.getExpiration());
             } catch (IOException ie) {
                 System.out.println("Caught IOException !");
             }
        } catch (MalformedURLException e) {
            System.out.println("Caught MUE !");
        }
    }
}
```

When this program is compiled (javac URLConnTest.java) and run (java URLConnTest), the following text is printed on the screen.

```
URL: http://cosmos/~asriniva/index.html
Content-Type: text/html
Content-Length: 443
Content-Encoding: null
```

```
Date (value): 829693487000 (string): Tue Apr 16 18:24:47 EDT 1996
Last modified on (value): 829686465000 (string): Tue Apr 16 16:27:45
EDT 1996
Expires on: 0
```

## getContentLength()

**ClassName**

URLConnection

**Purpose**

Gets the length of the content of the remote object that this URLConnection is connected to.

**Syntax**

public int getContentLength()

**Parameters**

None.

**Description**

One of the MIME header fields that specifies information about the object is the Content-Length field. It implies that the object data should be treated as binary data, and the integer value associated with this field specifies the data size of the remote object.

**Imports**

*import java.net.URLConnection;*

**Returns**

The integer value returned by this method specifies the size of the content of the remote object. If the content length cannot be determined, this method returns the value -1.

**Example**

This method is used in the example for the getContentEncoding method of this class.

## getContentType()

**ClassName**

URLConnection

**Purpose**

Gets the MIME-type/subtype combination of the remote object that this URLConnection is connected to.

**Syntax**

public String getContentType()

**Parameters**

None.

**Description**

The value associated with the Content-Type field of the MIME header specifies the MIME-type and subtype of the object that the URL refers to.

**Imports**

*import java.net.URLConnection;*

**Returns**

The String returned by this method contains the MIME-type and subtype combination of the remote object. If the content type is not known, this method returns a null value. For example, if the URL referred to a HTML file, this method would return text/html, whereas if the URL referred to a postscript file, the string returned by this method would be application/postscript.

**Example**

This method is used in the example for the getContentEncoding method of this class.

## getDate()

**ClassName**

URLConnection

**Purpose**

Gets the date and time that the object was sent.

**Syntax**

public long getDate()

**Parameters**

None.

**Description**

One of the MIME header fields is the Date field. The value associated with this field specifies the date and time that the data, accompanying this header, was sent. The Date field of the MIME header contains a string representation of the date, such as the following text.

Mon, 15 Apr 1996 09:00:00 GMT

This method invokes the parse method of the Date class to convert this string representation into a time value, and returns this time value.

**Imports**

*import java.net.URLConnection;*

**Returns**

The value returned by this method is obtained by invoking the parse method of the Date class which returns the number of milliseconds since the beginning epoch, for the specified date. If the date is not known, the value 0 is returned.

**See Also**

The parse method of the Date class. This class is described in Chapter 13.

**Example**

This method is used in the example for the getContentEncoding method of this class. The Date class is used to convert the number returned by this method to a string representation of a date and time.

## getDefaultAllowUserInteraction()

**ClassName**

URLConnection

**Purpose**

Gets the default value of the flag that indicates whether the protocol permits user-interaction while establishing the connection to the remote object.

**Syntax**

public static boolean getDefaultAllowUserInteraction()

**Parameters**

None.

**Description**

This method can be used to determine whether or not the protocol permits user-interaction. This property is required of some protocols that require user-interaction, such as typing in a password, before proceeding further. The variable that stores the value of the flag associated with this property is a static variable in the URLConnection class. This ensures uniformity in this policy among all instances of the URLConnection class for that protocol.

**Imports**

*java.net.URLConnection;*

**Returns**

The return value is true or false, depending on whether or not the default value associated with this property for this protocol is true or false. The default value for this variable is false.

**See Also**

The setDefaultAllowUserInteraction method; the getAllowUserInteraction and the setAllowUserInteraction methods of this class

**Example**

This method is used in the example for the getAllowUserInteraction method of this class.

## getDefaultRequestProperty(String)

**ClassName**

URLConnection

**Purpose**

Gets the default value associated with the specified field of the request header.

**Syntax**

public static String getDefaultRequestProperty(String key)

**Parameters**

***key***

The name of the request header field for which the default value is to be returned.

**Description**

In a protocol transaction such as http, the client sends a list of fields to the http server. Some of these fields (such as Accept and Accept Encoding) convey information about the capabilities of the client. This method is used to determine the default values associated with a specified field of the request header. The request fields (properties) contained in a HTTP protocol header are

From Accept Accept-Encoding

| | | |
|---|---|---|
| Accept-Language | User-Agent | Referrer |
| Authorization | Charge-to | If-Modified-Since |
| Pragma | | |

**Imports**

*import java.net.URLConnection;*

**Returns**

This method returns a String object that contains a default value associated with the specified property. The default implementation of this method (in the URLConnection class) simply returns a null string.

**See Also**

The setDefaultRequestProperty method of this class

**Example**

The following example shows how you can override this method.

```
// File: CustomURLConnection.java
//        URLConnection object for a new protocol
package CustomProtocolConnection;
import java.net.*;

public class CustomURLConnection extends URLConnection {
        // code for the URLConnection class methods that implement
        the specifics of the
        // protocol is written here
        ....
        public static String getDefaultRequestProperty(String key) {
               ....
               // return the default string for the specified key
               ....
        }
        ...
}
```





## getDefaultUseCaches()

**ClassName**

URLConnection

**Purpose**

Gets the default value of the flag that indicates whether the protocol should use the cache to retrieve an object or whether it should ignore the cache and fetch the object from the remote site.

**Syntax**

public boolean getDefaultUseCaches()

**Parameters**

None.

**Description**

Some protocols use local caches to enable quick access to previously retrieved objects. The URLConnection class maintains a per-instance boolean variable that indicates whether or not to use caching. A static boolean variable specifies the default value for the per-instance variable. This method returns the value of the static boolean variable.

**Imports**

*import java.net.URLConnection;*

**Returns**

The return value is true or false, depending on whether the URLConnection should use the cache or ignore the cache. The default value for this variable is true.

**See Also**

The setDefaultUseCaches method the setUseCaches and getUseCaches methods of this class

**Example**

This method is used in the example for the getAllowUserInteraction method of this class.

## getDoInput()

**ClassName**

URLConnection

**Purpose**

Gets the value of the flag that indicates whether this URLConnection can be used for input (i.e. can be read from).

**Syntax**

public boolean getDoInput()

**Parameters**

None.

**Description**

The value returned by this method indicates whether or not data can be read from the URLConnection.

**Imports**

*import java.net.URLConnection;*

**Returns**

The boolean value returned is true if the URLConnection can be used for input and false if not. The default value for this flag is true.

**See Also**

The setDoInput method of this class

**Example**

This method is used in the example for the getAllowUserInteraction method of this class.

## getDoOutput()

**ClassName**

URLConnection

**Purpose**

Gets the value of the flag that indicates whether this URLConnection can be used for output (i.e., can be written to).

**Syntax**

public boolean getDoOutput()

**Parameters**

None.

**Description**

The value returned by this method indicates whether or not data can be written to the remote object that the URLConnection is connected to.

**Imports**

*import java.net.URLConnection;*

**Returns**

The boolean value returned is true if the URLConnection can be used for output and false if not. The default value for this flag is false.

**See Also**

The setDoOutput method of this class

**Example**

This method is used in the example for the getAllowUserInteraction method of this class.

## getExpiration()

**ClassName**

URLConnection

**Purpose**

Gets the date after which the information retrieved from the remote object ceases to be valid and must be reloaded.

**Syntax**

public long getExpiration()

**Parameters**

None.

**Description**

One of the MIME header fields that specifies information about the object is the

Expires field. The value associated with this field specifies the date after which the information already retrieved ceases to be valid and must be retrieved again. The object referred to by the URL is considered to be stale after the date/time specified in this field. This allows the data to be refreshed periodically and also allows a limited amount of control over caching policies. The Expires field of the MIME header contains a string representation of the date, such as the following text.

Mon, 15 Apr 1996 09:00:00 GMT

This method invokes the parse method of the Date class to convert this string representation into a time value, and returns this time value.

**Imports**

*import java.net.URLConnection;*

**Returns**

The return value indicates the date after which the object needs to be retrieved again. The value returned by this method is obtained by invoking the parse method of the Date class that returns the number of milliseconds since the beginning epoch, for the specified date. If the expiration date is not known then the value 0 is returned.

**See Also**

The parse method of the Date class. This class is described in Chapter 13.

**Example**

This usage of this method is illustrated in the example for the getContentEncoding method of this class.

## getHeaderField(int)

**ClassName**

URLConnection

**Purpose**

Gets the value associated with the nth header field of the object's header.

**Syntax**

public String getHeaderField(int n)

**Parameters**

***n***

The index number of the header field whose value is to be retrieved.

**Description**

This method retrieves the value associated with a particular header field. The header field is specified using an index. Without knowing the actual field names contained in the header, one can use this method and the GetHeaderFieldKey method to iterate through all the header field names and associated values. The index n starts at zero.

**Imports**

*import java.net.URLConnection;*

**Returns**

If the index n is less than the number of fields in the header, then the value for that field is returned. If n is greater than the number of fields in the header, the

return value is null. The default implementation of this method (in the URLConnection class) simply returns a null string.

**See Also**

The URLConnection.getHeaderField(String) method

**Example**

Please refer to the example for the getDefaultRequestProperty method of this class to see how you can provide a custom implementation of this method in any class that extends the URLConnection class.

## getHeaderField(String)

**ClassName**

URLConnection

**Purpose**

Gets the value associated with the specified header field.

**Syntax**

public String getHeaderField(String name)

**Parameters**

***name***

The name of the header field whose value is to be retrieved.

**Description**

This method retrieves the value associated with a specified header field. The header field is specified by name (e.g,. Content-Type).

**Imports**

*import java.net.URLConnection;*

**Returns**

The String object returned by this method contains the value associated with the specified field name. If the header field name was not found in the data stream read from the remote object, then the value returned by this method is null. The default implementation of this method (in the URLConnection class) simply returns a null string. The header fields given with or in relation to objects in HTTP are

| | | |
|---|---|---|
| Allowed | Public | Content-Length |
| Content-Type | Content-Transfer-Encoding | Content-Encoding |
| Date | Expires | Last-Modified |
| Message-Id | URL | Version |
| Derived-From | Content-Language | Cost |
| Link | Title | |

**See Also**

The getHeaderField method of this class

**Example**

Please refer to the example for the getDefaultRequestProperty method of this class to see how you can provide a custom implementation of this method in any class that extends the URLConnection class.

## getHeaderFieldDate(String, long)

**ClassName**

URLConnection

**Purpose**

Gets the value associated with the specified header field, reads this value as a date value, and returns the date value.

**Syntax**

public long getHeaderFieldDate(String name, long strDef)

**Parameters**

***name***

The name of the header field whose value is to be retrieved as a date value.

***strDef***

The default value to return if the header field *name* is not found.

**Description**

This method reads the value associated with the header field and returns this value as a date value. This method can be used by protocols that have pre-parsed headers where the formats of individual fields are known. Please refer to the getDate method description for the format of the date value returned by this method.

**Imports**

*import java.net.URLConnection;*

**Returns**

This method returns the value specified in *strDef* if the field name is not known. If the field name is found, the value associated with it is returned as a long integer.

**Example**

This helper method, shown below, can be used to extract the contents of a field and convert it into a long integer representing the number of milliseconds since the epoch.

```
// File: CustomURLConnection.java
//        URLConnection object for a new protocol
package CustomProtocolConnection;
import java.net.*;

public class CustomURLConnection extends URLConnection {
        // code for the URLConnection class methods that implement
        the specifics of the
        // protocol is written here
        ....
        // assume that this protocol has a header field which
        specifies the local time/date
        public long getLocalTime() {
                return getHeaderFieldDate("local-time", 0);
        }
        ...
```

```
}
```

## getHeaderFieldInt(String, int)

**ClassName**

URLConnection

**Purpose**

Gets the value associated with the specified header field, reads this value as an integer value, and returns the integer value.

**Syntax**

public int getHeaderFieldInt(String name, int valDef)

**Parameters**

***name***

The name of the header field whose value is to be retrieved as an integer value.

***valDef***

The default value to return if the header field *name* is not found.

**Description**

This method reads the value associated with the header field and returns this value as an integer value. This method can be used by protocols that have pre-parsed headers, where the formats of individual fields are known.

**Imports**

*java.net.URLConnection;*

**Returns**

This method returns the value specified in *valDef* if the field name is not known. If the field name is found, the value associated with it is returned as an integer.

**Example**

This helper method can be used in a manner similar to that described in the example for the getHeaderFieldDate method description.

## getHeaderFieldKey(int)

**ClassName**

URLConnection

**Purpose**

Gets the field name of the nth header field.

**Syntax**

public String getHeaderFieldKey(int n)

**Parameters**

***n***

The index number of the header field whose field name is to be retrieved.

**Description**

This method retrieves the field name associated with a particular header field. The header field is specified using an index. Without knowing the actual field names contained in the header, one can use this method and the GetHeaderField (int) method to iterate through all the header field names and their associated values. The index *n* starts at zero.

**Imports**

*import java.net.URLConnection;*

**Returns**

If index *n* is less than the number of fields in the header, then the field name for that field is returned. If *n* is greater than the number of fields in the header, the return value is null. The default implementation of this method (in the URLConnection class) simply returns a null string.

**See Also**

The getHeader method of this class

**Example**

Please refer to the example for the getDefaultRequestProperty method of this class to see how you can provide a custom implementation of this method in any class that extends the URLConnection class.

## getIfModifiedSince()

**ClassName**

URLConnection

**Purpose**

Gets the time that is sent as the value of the If-Modified-Since header field of the request header to determine whether or not an object should be retrieved.

**Syntax**

public long getIfModifiedSince()

**Parameters**

None.

**Description**

The If-Modified-Since field of the request header is sent by the client to the server in order to make the retrieval of an object conditional. By specifying a time with this header field, the client instructs the server not to send the object, if the object has not changed since the time indicated, as the value for this header field. This method gets the value associated with this field.

**Imports**

*import java.net.URLConnection;*

**Returns**

The return value is a long integer that indicates the time that should be used by a server to determine whether an object has been modified or not.

**Example**

This method can be used in a manner similar to the getLastModified method.

## getInputStream()

**ClassName**

URLConnection

**Purpose**

Gets an input stream from which data from the remote object can be read.

**Syntax**

public InputStream getInputStream() throws IOException

**Parameters**

None.

**Description**

This method opens and returns an InputStream to the remote object that this URLConnection object is connected to. The object's data can be read using the InputStream object returned by this method. Protocols that permit input must implement this method.

**Imports**

*import java.net.URLConnection;*

**Returns**

This method returns an InputStream object that can be used to read data from the remote object. An UnknownServiceException is generated if the protocol does not support input. The default implementation of this method (in the URLConnection class) simply throws an UnknownServiceException.

**Example**

This method is used in the example for the openConnection method of the URL class.

## getLastModified()

**ClassName**

URLConnection

**Purpose**

Gets the time that the object that this URLConnection is connected to was last modified.

**Syntax**

public long getLastModified()

**Parameters**

None.

**Description**

One of the MIME header fields that specifies information about the object is the Last-Modified field. The value associated with this field specifies the time that the object was last modified. The format of this field is the same as the date format for the Date field described in the getDate method of this class. The server compares this value against the value of the If-Modified-Since field of the request header to determine whether the object needs to be sent to the client.

**Imports**

*import java.net.URLConnection;*

**Returns**

The return value indicates the time that the object this URLConnection is connected to was last modified. If the time is not known, the value 0 is returned.

**See Also**

The getIfModifiedSince method of this class; the parse method of the Date class, and the Date class, described in Chapter 13.

**Example**

An example of this method is illustrated in the example for the getContentEncoding method of this class.

## getOutputStream()

**ClassName**

URLConnection

**Purpose**

Gets an output stream that the remote object can be written to.

**Syntax**

public OutputStream getOutputStream() throws IOException

**Parameters**

None.

**Description**

This method opens and returns an OutputStream to the remote object that this URLConnection object is connected to. Data can be written to the object using the OutputStream object returned by this method. Writing to a URLConnection is similar to the POST method of the HTTP protocol. It provides a means by which information can be sent to the object referenced by the URL. You can use the output stream of a URLConnection to send data back to CGI (Common Gateway Interface) scripts that are running on a World Wide Web server. Protocols that permit output must implement this method.

**Imports**

*import java.net.URLConnection;*

**Returns**

This method returns an OutputStream object. An UnknownServiceException is generated if the protocol does not support output. The default implementation of this method (in the URLConnection class) simply throws an UnknownServiceException.

**Example**

The following example is a simple template that you can use to write Java programs that post and receive data to or from CGI scripts.

```
// File: URLConnWrite.java
//       A template for interacting with CGI scripts

import java.net.*;
import java.io.*;

public class URLConnWrite {
    public static void main(String args[]) {
                try {
                                // This URL points to a test cgi
                                script that simply
                                // echoes the data posted to it, along
                                with a list
                                // of environment variables
```

```
                    URL cgiURL = new URL("http://
                    cosmos/cgi-bin/test-cgi.tcl");
                    try {
                                // get a handle to the
                                URLConnection object
                                // for this URL
                                URLConnection uc =
                                cgiURL.openConnection();

                                // Attach an output stream
                                and write to this stream
                                OutputStream os =
                                uc.getOutputStream();
                                PrintStream ps = new
                                PrintStream(os);
                                String data;
                                // Fill this string with
                                contain the data to be sent
                                data = new String("Java
                                AND book AND Waite");

                                // encode the string prior
                                to sending it to the server
                                data =
URLEncoder.encode(data);
                                ps.println("query
                                =" + data);    // send
                                the string
                                ps.close();   // close
                                the stream
                                // now read the response
                                from the CGI script
                                InputStream is =
                                uc.getInputStream();
                                DataInputStream di = new
                                DataInputStream(is);
                                String line;
                                int lineNumber = 1;
                                // This example simply
                                prints the response on
                                the screen
                                while ((line =
                                di.readLine()) != null) {
                                 System.out.println
                                 ("Line " + lineNumber
                                 + ": " + line);
                                 lineNumber++;
                                }
                                di.close();   // close
                                the input stream
                    } catch (IOException ie) {
                        System.out.println("Caught IOE !");
                    }
        } catch (MalformedURLException e) {
            System.out.println("Caught MUE !");
        }
    }
```

```
}
```

The output of this program is:

```
Line 1: <HTML>
Line 2: <HEAD>
Line 3: <TITLE>CGI/1.0 TCL script report:</TITLE>
Line 4: </HEAD>
Line 5: <BODY>
Line 6: <H1>Command Line Arguments</H1>
Line 7: argc is 0. argv is .
Line 8:
Line 9: <H1>Message</H1>
Line 10: <PRE>
Line 11: query = Java AND book AND Waite
Line 12:
Line 13: </PRE>
Line 14: <H1>Environment Variables</H1>
Line 15: <DL>
Line 16: <DT>SERVER_SOFTWARE<DD>NCSA/1.3
Line 17: <DT>SERVER_NAME<DD>199.100.97.2
Line 18: <DT>GATEWAY_INTERFACE<DD>CGI/1.1
Line 19: <DT>SERVER_PROTOCOL<DD>HTTP/1.0
Line 20: <DT>SERVER_PORT<DD>80
Line 21: <DT>REQUEST_METHOD<DD>POST
Line 22: <DT>SCRIPT_NAME<DD>/cgi-bin/test-cgi.tcl
Line 23: <DT>QUERY_STRING<DD>
Line 24: <DT>REMOTE_HOST<DD>cosmos
Line 25: <DT>REMOTE_ADDR<DD>199.100.97.2
Line 26: <DT>CONTENT_TYPE<DD>application/x-www-form-urlencoded
Line 27: <DT>CONTENT_LENGTH<DD>30
Line 28: <DT>HTTP_ACCEPT<DD>
Line 29: */*;
Line 30: *;
Line 31: image/gif,
Line 32: image/jpeg,
Line 33: q=.2
Line 34: q=.2,
Line 35: text/html,
Line 36: </DL>
Line 37: </BODY>
Line 38: </HTML>
```

## getRequestProperty()

**ClassName**

URLConnection

**Purpose**

Gets the value associated with the specified request header field name.

**Syntax**

public String getRequestProperty(String key)

**Parameters**

***key***

The request header field name whose associated value is to be returned

**Description**

In a protocol transaction such as http, the client sends a list of fields to the http server. These fields are part of the request header. In the http protocol, From, Accept, If-Modified-Since, and Pragma are some of the field names sent in the request header. A more comprehensive list of property names is listed in the getDefaultRequestProperty method description. This method is used to retrieve the value associated with a specific property. This method should not be invoked if the URLConnection object is already "connect()"ed.

**Imports**

*import java.net.URLConnection;*

**Returns**

The String returned by this method contains the value of the request header field. If the header field is not found, then the null value is returned. The default implementation of this method (in the URLConnection class) throws an IllegalAccessError if the URLConnection object is already connected; otherwise it returns a null string.

**See Also**

The setRequestProperty method of this class

**Example**

Please refer to the example for the getDefaultRequestProperty method of this class to see how you can provide a custom implementation of this method in any class that extends the URLConnection class.

## getURL()

**ClassName**

URLConnection

**Purpose**

Gets the URL object to which this connection was established.

**Syntax**

public URL getURL()

**Parameters**

None.

**Description**

This method simply returns the URL object that was supplied as a parameter to the URLConnection constructor.

**Imports**

*import java.net.URLConnection;*

**Returns**

This method returns the URL object that this URLConnection established a connection to.

**Example**

This usage of this method is illustrated in the following example.

```
// File: URLConnTest.java
//       Extracts and prints the URL string specification associated
with
```

```
//       an URLConnection object
import java.net.*;
import java.io.IOException;

public class URLConnTest {
    public static void main(String args[]) {
        try {
       // Replace the following URL with any valid URL
       URL u1 = new URL("http://cosmos/~asriniva/index.html");
            try {
                URLConnection uc = u1.openConnection();
                System.out.println("URL.toString: " + u1.toString());
                System.out.println("URLConnection.getURL().toString: "
+
                                   uc.getURL().toString());
            } catch (IOException ie) {
                System.out.println("Caught IOException !");
            }
        } catch (MalformedURLException e) {
            System.out.println("Caught MUE !");
        }
    }
}
```

## getUseCaches()

**ClassName**

URLConnection

**Purpose**

Gets the value of the flag that indicates whether this URLConnection object should use the cache to retrieve an object or whether it should ignore the cache and fetch the object from the remote site.

**Syntax**

public boolean getUseCaches()

**Parameters**

None.

**Description**

Some protocols use local caches to enable quick access to previously retrieved objects. The URLConnection class maintains a per-instance boolean variable that indicates whether or not to use caching. This method returns the value of this boolean variable.

**Imports**

*import java.net.URLConnection;*

**Returns**

The return value is true or false, depending on whether the URLConnection should use the cache or ignore the cache. The default value returned by this method is true.

**See Also**

The setUseCaches method; the setDefaultUseCaches method and getDefaultUseCaches methods of this class

**Example**

This method is used in the example for the getAllowUserInteraction method of this class.

## guessContentTypeFromName(String)

**ClassName**

URLConnection

**Purpose**

Guesses the MIME-type/subtype of the specified object by examining its name.

**Syntax**

static protected String guessContentTypeFromName(String fname)

**Parameters**

***fname***

The specification of the remote object (can be a text representation of a URL or just a plain file name).

**Description**

This method tries to determine the MIME-type of the specified object,by examining the file name extension of the object. The file name extension is compared against a list of extensions for which the MIME-type/subtype combinations are known. For example, if *fname* was http://www.syr.edu/smiley.gif, then the .gif extension would have been identified as a known MIME-type and would have returned the string image/gif. The following is a list of file name extensions and the corresponding MIME-type/subtype combinations that this method returns.

| **Extension** | **MIME-type/subtype** |
|---|---|
| <None> | content/unknown |
| .uu | application/octet-stream |
| .saveme | application/octet-stream |
| .dump | application/octet-stream |
| .hqx | application/octet-stream |
| arc | application/octet-stream |
| .o | application/octet-stream |
| .a | application/octet-stream |
| .bin | application/octet-stream |
| .exe | application/octet-stream |
| .z | application/octet-stream |
| .gz | application/octet-stream |
| .oda | application/oda |
| .pdf | application/pdf |
| .eps | application/postscript |
| .ai | application/postscript |
| .ps | application/postscript |
| .rtf | application/rtf |
| .dvi | application/x-dvi |
| .hdf | application/x-hdf |
| .latex | application/x-latex |
| .cdf | application/x-netcdf |
| .nc | application/x-netcdf |

```
.tex                    application/x-tex
.texinfo                application/x-texinfo
.texi                   application/x-texinfo
.t                      application/x-troff
.tr                     application/x-troff
.roff                   application/x-troff
.man                    application/x-troff-man
.me                     application/x-troff-me
.ms                     application/x-troff-ms
.src                    application/x-wais-source
.wsrc                   application/x-wais-source
.zip                    application/zip
.bcpio                  application/x-bcpio
.cpio                   application/x-cpio
.gtar                   application/x-gtar
.shar                   application/x-shar
.sh                     application/x-shar
.sv4cpio                application/x-sv4cpio
.sv4crc                 application/x-sv4crc
.tar                    application/x-tar
.ustar                  application/x-ustar
.snd                    audio/basic
.au                     audio/basic
.aifc                   audio/x-aiff
.aif                    audio/x-aiff
.aiff                   audio/x-aiff
.wav                    audio/x-wav
.gif                    image/gif
.ief                    image/ief
.jfif                   image/jpeg
.jfif-tbnl              image/jpeg
.jpe                    image/jpeg
.jpg                    image/jpeg
.jpeg                   image/jpeg
.tif                    image/tiff
.tiff                   image/tiff
.ras                    image/x-cmu-rast
.pnm                    image/x-portable-anymap
X.pbm                    image/x-portable-bitmap
.pgm                    image/x-portable-graymap
.ppm                    image/x-portable-pixmap
.rgb                    image/x-rgb
.xbm                    image/x-xbitmap
.xpm                    image/x-xpixmap
.xwd                    image/x-xwindowdump
.htm                    text/html
.html                   text/html
.text                   text/plain
.c                      text/plain
.cc                     text/plain
.c++                    text/plain
.h                      text/plain
.pl                     text/plain
.txt                    text/plain
.java                   text/plain
.rtx                    application/rtf
.tsv                    text/tab-separated-values
```

```
    .etx                 text/x-setext
    .mpg                 video/mpeg
    .mpe                 video/mpeg
    .mpeg                video/mpeg
    .mov                 video/quicktime
    .qt                  video/quicktime
    .avi                 application/x-troff-msvideo
    .movie               video/x-sgi-movie
    .mv                  video/x-sgi-movie
    .mime                message/rfc822
```

**Imports**

*import java.net.URLConnection;*

**Returns**

If the extension of the specified object matched a known extension, then the corresponding MIME-type/subtype combination is returned. The string content/unknown is returned for file names whose extensions do not match any of the known extensions.

**See Also**

The guessContentTypeFromStream method of this class

**Example**

This protected method can only be accessed by subclasses of the URLConnection class and by classes in the *java.net* package. The following code sample shows how you might invoke this method within a class that extends the URLConnection class.

```
// File: CustomURLConnection.java
//        URLConnection object for a new protocol
package CustomProtocolConnection;
import java.net.*;

public class CustomURLConnection extends URLConnection {
   // code for the URLConnection class methods that implement the
   specifics of the
   // protocol is written here
   ....
   public void connnect() throws IOException {
           ....
           String urlSpec = url.toString();
           String contentType = guessContentTypeFromName(urlSpec);
           // Take the necessary action, depending on the content type
           ....
   }
   ...
}
```

## guessContentTypeFromStream(InputStream)

**ClassName**

URLConnection

**Purpose**

Guesses the MIME-type/subtype of the specified object by inspecting the data read from the specified InputStream.

**Syntax**

static protected String guessContentTypeFromStream(InputStream is) throws IOException

**Parameters**

***is***

The InputStream that is connected to the remote object whose content type is to be determined

**Description**

This method tries to determine the content type of an object by reading and examining data from the object. Many content types such as image files have magic strings as part of the header of the object that identify the object as a particular image format. Use this method with care!

**Imports**

*import java.net.URLConnection;*

**Returns**

If a valid content type is detected, this method returns the MIME-type/subtype combination of the object; otherwise, it returns a null value.

**See Also**

The guessContentTypeFromName method of this class

**Example**

This protected method can only be accessed by subclasses of the URLConnection class and by classes in the *java.net* package. This method can be used in a manner similar to the way that the guessContentTypeFromName is used. The difference between the two is that the parameter supplied to this method is an InputStream object instead of a file name.

## setAllowUserInteraction(boolean)

**ClassName**

URLConnection

**Purpose**

Sets the value of the flag that indicates whether the protocol permits user-interaction while establishing the connection to the remote object.

**Syntax**

public void setAllowUserInteraction(boolean bFlag)

**Parameters**

***bFlag***

The boolean value that, if set to true, allows user-interaction or, if set to false, disables user-interaction.

**Description**

Protocols such as http, that have access and security control features, allow user-interaction (such as authentication by asking for a user name and password) during the process of setting up a connection to an object referred to in a URL. Protocol implementors must specify whether this interaction is permitted by the

protocol or not. This variable is maintained on a per-instance basis for the URLConnection class. This method cannot be invoked if the URLConnection object is already “connect()”ed.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**See Also**

The getAllowUserInteraction, getDefaultAllowUserInteraction, and the setDefaultAllowUserInteraction methods of this class

**Example**

The following example illustrates how you might invoke this method to manipulate whether or not to allow user-interaction.

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

public class URLConnTest1 {
    public static void main(String args[]) {
        try {
       // Replace the following URL with any valid URL
       URL u1 = new URL("http://cosmos/~asriniva/index.html");
            try {
                URLConnection uc = u1.openConnection();
                System.out.println(uc.getAllowUserInteraction());
                uc.setAllowUserInteraction(true);
                System.out.println(uc.getAllowUserInteraction());
            } catch (IOException ie) {
                System.out.println("Caught IOException !");
            }
        } catch (MalformedURLException e) {
            System.out.println("Caught MUE !");
        }
   }
}
```

## setContentHandlerFactory(ContentHandlerFactory)

**ClassName**

URLConnection

**Purpose**

Specifies the factory object (that implements the ContentHandlerFactory interface) that knows how to create content-specific ContentHandler objects.

**Syntax**

public static synchronized void setContentHandlerFactory(ContentHandlerFactory factory)

**Parameters**

***factory***

The ContentHandlerFactory object that must be used to create content-specific content handlers.

**Description**

All objects of the URLConnection class share the same ContentHandlerFactory object. By invoking this method, you can install your own ContentHandlerFactory. An error is generated if a ContentHandlerFactory already exists. If you write new content handler classes and install these classes in nonstandard locations, then you will need to create a class that implements the ContentHandlerFactory interface. This class will need to know where to find the content-specific implementations of the ContentHandler class.

**Imports**

*java.net.URLConnection;*

**Returns**

None.

**See Also**

The ContentHandlerFactory class described in this chapter

**Example**

The following code sample shows how you can direct the URLConnection class to use a custom ContentHandlerFactory object.

```
import java.net.*;
import CustomPackage.CustomContentHandlerFactory;

        .....
        .....
        // in the context of a method in your Java application
        CustomContentHandlerFactory f = new CustomContentHandler
        Factory();
        URLConnection.setContentHandlerFactory(f);
        // henceforth, all URLConnections will use this new factory
        object to
        // create content handlers
        .....
        .....
```

## setDefaultAllowUserInteraction(boolean)

**ClassName**

URLConnection

**Purpose**

Sets the default value of the flag that indicates whether the protocol permits user-interaction while establishing the connection to the remote object.

**Syntax**

public static void setDefaultAllowUserInteraction(boolean bFlag)

**Parameters**

***bFlag***

The default boolean value for this property.

**Description**

The variable that stores the value of the flag associated with this property is a static variable in the URLConnection class. Setting this variable will affect all future connections made by URLConnection classes created henceforth.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**See Also**

The getDefaultAllowUserInteraction, getAllowUserInteraction, and setAllowUserInteraction methods of this class

**Example**

This method is a static method of the URLConnection class and does not need an object reference to invoke the method. You can invoke the method by prefixing it with the class name as shown here:

```
..... // somewhere in your Java application
URLConnection.setDefaultAllowUserInteraction(true);
.....
```

## setDefaultRequestProperty(String, String)

**ClassName**

URLConnection

**Purpose**

Sets the default value associated with the specified field of the request header.

**Syntax**

public static void setDefaultRequestProperty(String fieldName, String fieldValue)

**Parameters**

***fieldName***

The name of the field in the request header.

***fieldValue***

The default value to assign to the above field.

**Description**

In a protocol transaction such as http, the client sends a list of fields to the http server. Some of these fields (such as Accept and Accept Encoding) convey information about the capabilities of the client. This method is used to set the default values associated with a specified field of the request header. For example, you may set the default value of the Accept field to text/plain, text/html, and image/gif. Doing this allows the client to inform the server that it can accept plain files and HTML files, as well as GIF images. The values set here are used for initialization whenever a URLConnection object is constructed. The default implementation of this method (in the URLConnection class) does nothing.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**See Also**

The getDefaultRequestProperty method of this class

## setDefaultUseCaches(boolean)

**ClassName**

URLConnection

**Purpose**

Sets the default value of the flag that indicates whether the protocol should use the cache to retrieve an object or whether it should ignore the cache and fetch the object from the remote site.

**Syntax**

public void setDefaultUseCaches(boolean bFlag)

**Parameters**

***bFlag***

The default boolean value for this property.

**Description**

Some protocols use local caches to enable quick access to previously retrieved objects. The URLConnection class maintains a per-instance boolean variable that indicates whether to use caching or not. A static boolean variable specifies the default value for the per-instance variable. This method sets the value of the static boolean variable. Setting this value affects all URLConnection objects created hereafter.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**See Also**

The getDefaultUseCaches, setUseCaches, and getUseCaches methods of this class

## setDoInput(boolean)

**ClassName**

URLConnection

**Purpose**

Sets the value of the flag that indicates whether this URLConnection can be used for input (i.e., can be read from).

**Syntax**

public void setDoInput(boolean bFlag)

**Parameters**

***bFlag***

The boolean value for this property. If set to true, this URLConnection object can be read from.

**Description**

The parameter *bFlag* specifies whether or not this URLConnection object can be used for input. If it is set to false, data cannot be read from this URLConnection object. This method should not be invoked if the URLConnection object is already “connect()”ed.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**See Also**

The getDoInput method of this class

## setDoOutput(boolean)

**ClassName**

URLConnection

**Purpose**

Sets the value of the flag that indicates whether this URLConnection can be used for output (i.e., can be written to).

**Syntax**

public void setDoOutput(boolean bFlag)

**Parameters**

***bFlag***

The boolean value for this property. If set to true, this URLConnection object can be used for output (i.e, it can be written to).

**Description**

The parameter *bFlag* specifies whether or not this URLConnection object can be used for output. If it is set to false, data cannot be written to the remote object using this URLConnection object. This method should not be invoked if the URLConnection object is already "connect()"ed.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**See Also**

The getDoOutput method of this class

## setIfModifiedSince(long)

**ClassName**

URLConnection

**Purpose**

Sets the time sent as the value of the If-Modified-Since header field of the request header to determine whether or not an object should be retrieved.

**Syntax**

public void setIfModifiedSince(long timeValue)

**Parameters**

***timeValue***

The time value that the server should use for comparison to determine whether or not the object is to be sent to this URLConnection.

**Description**

The If-Modified-Since field of the request header is sent by the client to the server in order to make the retrieval of an object conditional. By specifying a time with this header field, the client instructs the server not to send the object if the object

has not changed since the time indicated by the value for this header field. This method sets the value associated with this field. This method should not be invoked if the URLConnection object is already “connect()”ed.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

## setRequestProperty(String, String)

**ClassName**

URLConnection

**Purpose**

Sets the value associated with the specified request header field name.

**Syntax**

public void setRequestProperty(String key, String value)

**Parameters**

***key***

The name of the field in the request header.

***value***

The default value to assign to the above field.

**Description**

In a protocol transaction such as http, the client sends a list of fields to the http server. These fields are part of the request header. In the http protocol, From, Accept, If-Modified-Since, and Pragma are some of the field names sent in the request header. This method is used to set the value associated with a specific property. This method should not be invoked if the URLConnection object is already “connect()”ed. The default implementation of this method (in the URLConnection class) throws an IllegalAccessError if the URLConnection object is already “connect()”ed.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**See Also**

The getRequestProperty method of this class

## setUseCaches(boolean)

**ClassName**

URLConnection

**Purpose**

Sets the value of the flag that indicates whether this URLConnection object should use the cache to retrieve an object or whether it should ignore the cache and fetch the object from the remote site.

**Syntax**

public void setUseCaches(boolean bFlag)

**Parameters**

***bFlag***

The boolean value that indicates whether this URLConnection should use the caches or not.

**Description**

Some protocols use local caches to enable quick access to previously retrieved objects. The URLConnection class maintains a per-instance boolean variable that indicates whether to use caching or not. This method sets the value of this boolean variable. This method should not be invoked if the URLConnection object is already "connect()"ed.

**Imports**

*import java.net.URLConnection;*

**Returns**

None.

**See Also**

The getUseCaches, setDefaultUseCaches, and getDefaultUseCaches methods of this class

## toString()

**ClassName**

URLConnection

**Purpose**

Represents the parameters of the URLConnection object as a String.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method is used when the parameters associated with this URLConnection object are to be printed as a string. This method is generally used for debugging purposes.

**Imports**

*import java.net.URLConnection;*

**Returns**

This method returns a String object that contains the values of the parameters for this URLConnection object. The default implementation of this method prints the class name and the URL that this URLConnection object was created for. Subclasses of the URLConnection class can override this method to provide more information.

**Example**

The following example prints the text representation of a URLConnection object.

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

public class URLConnPrint {
    public static void main(String args[]) {
        try {
       // Replace the following URL with any valid URL
       URL u1 = new URL("http://cosmos/~asriniva/index.html");
            try {
                URLConnection uc = u1.openConnection();
                System.out.println(uc.toString());
            } catch (IOException ie) {
                System.out.println("Caught IOException !");
            }
        } catch (MalformedURLException e) {
            System.out.println("Caught MUE !");
        }
    }
}
```

This example prints the following string on the screen:

```
sun.net.www.protocol.http.HttpURLConnection:http://cosmos/~asriniva/
index.html
```

## URLEncoder

**Purpose**

Encodes strings into URL format. Encoding a string in this uniform format ensures that the string is not corrupted by errors such as character set variations on different systems.

**Syntax**

public class URLEncoder extends Object

**Description**

Some protocols use characters (in the URL specification) that may cause problems due to corruption by imperfect gateways or to a difference in the character sets used in different environments. You will find this class very useful if you are writing Java programs that interact with CGI (Common Gateway Interface) scripts. Figure 11-7 shows the inheritance diagram for the URLEncoder class.

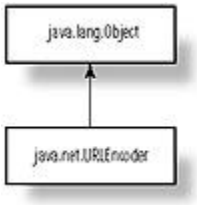


**Figure 11-7** Inheritance diagram for the URLEncoder class

**PackageName**

*java.net*

**Imports**

*import java.net.URLEncoder;*

**Constructors**

None.

**Parameters**

None.

**Example**

Refer to the example for the encode method of this class. This method is also used in the example for the getOutputStream method of the URLConnection class described in this chapter.

## encode(String)

**ClassName**

URLEncoder

**Purpose**

Returns a URL encoded form of the specified string.

**Syntax**

public static String encode(String s)

**Parameters**

***s***

The string to be encoded.

**Description**

This method simply translates the text string *s* into a URL encoded string and returns this string. The following (sets of) characters are left unchanged in the encoded form of the text string: characters A through Z, a through z, 0 through 9 and the underscore character (_). The space character () is replaced with the + sign. All other characters are replaced by a percent sign (%) followed by a 2-digit hexadecimal number. This hexadecimal number represents the value of the character.

**Imports**

*import java.net.URLEncoder;*

**Returns**

This method returns a String object that contains the URL encoded representation of the text in the string *s*.

**Example**

Because this method is static, you can invoke it simply by prefixing it with the class name (URLEncoder), as illustrated in the TestEncoder example shown here.

```
import java.net.URLEncoder;

public class TestEncoder {

   public static void main(String args[]) {
      String s1 = new String("http://myschool.edu/index.html");
      System.out.println("Source string: " + s1);
```

```
      System.out.println("URLEncoded version:" +
URLEncoder.encode(s1));
      System.out.println("");

      String s2 = new String("A text-string with 6_words");
      System.out.println("Source string: " + s2);
      System.out.println("URLEncoded version:" +
URLEncoder.encode(s2));
   }
}
```

When this example is compiled (javac TestEncoder.java) and run (java TestEncoder), it produces the following output.

```
Source string: http://myschool.edu/index.html
URLEncoded version:http%3a%2f%2fmyschool%2eedu%2findex%2ehtml
Source string: A text-string with 6_words
URLEncoded version:A+text%2dstring+with+6_words
```

## URLStreamHandler

**Purpose**

Specifies an abstract base class that must be subclassed to implement stream handlers for specific protocols (such as http, nntp, ftp, and so on).

**Syntax**

public class URLStreamHandler extends Object

**Description**

To implement a protocol handler in Java, you will need to subclass both the URLStreamHandler class and the URLConnection class. The URLStreamHandler object returns a URLConnection object, which is connnected to the specified URL. This URLConnection object implements the specifics of the protocol. This class defines the methods that can be overridden in the subclass to implement the protocol-specific functionality. Except for the constructor, the rest of the methods of this class are protected, which implies that these methods can be accessed only from within the package in which the subclasses are defined. An instance of the subclass that implements the functionality required for a specific protocol is created within the constructor of the URL class. Figure 11-2 shows the basic relationships between URLs, ProtocolHandlers, and ContentHandlers.

By convention, the URLStreamHandler class is always called Handler and it is always created by referring to the absolute path name for the class. The ambiguity of all the URLStreamHandler subclasses being called Handler is resolved by referring to each class using its absolute path name specification. The URL class, by default, looks for a class named Handler in a subdirectory named with the protocol name in the sun/net/www/protocol directory. The sun/net/www/protocol directory structure can be found along with the other standard directories containing the .class files that Java uses. For example, the subclass of the URLStreamHandler class that handles the http protocol will be located in the sun/net/www/protocol/http/Handler.class file. If you are writing protocol handlers, you can either follow this convention or you can write your own class that implements the URLStreamHandlerFactory interface. This factory should know how to create instances of your protocol handlers.

Figure 11-8 shows the inheritance diagram for the URLStreamHandler class.

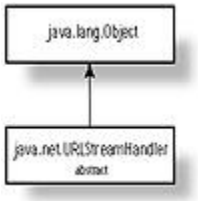


**Figure 11-8** Inheritance diagram for the URLStreamHandler class

**PackageName**

*java.net*

**Imports**

*import java.net.URLStreamHandler;*

**Constructors**

public URLStreamHandler()

**Parameters**

None.

**Example**

The following code shows how you create a custom URL StreamHandler.

```
// File: CustomURLConnection.java
//        URLConnection object for a new protocol
package CustomProtocolConnection;
import java.net.*;

public class CustomURLConnection extends URLConnection {
         // code for the URLConnection class methods that implement
         the specifics of the
         // protocol is written here
         ....
         public void connnect() throws IOException {
                  ....
         }
         ...
         public String getHeaderField(String name) {
                  ...
         }
         public InputStream getInputStream() throws IOException {
                  ...
         }
         ...
}
// File: Handler.java
//       This class will be referred to as sun.net.www.CustomProtocol.
Handler
package sun.net.www.protocol.CustomProtocol;
import java.net.*;
import CustormProtocolConnection.CustomURLConnection;

public class Handler extends URLStreamHandler {
         public URLConnection openConnection(URL u) throws
         IOException  {
                ....
```

```
                // create and return a CustomProtocolConnection
                object that
                // establishes a connection (as per the policies
                of the protocol) with the specified URL
                return new CustomURLConnection(u);
        }
}
```

## openConnection(URL)

**ClassName**

URLStreamHandler

**Purpose**

Opens an active connection to the specified URL and returns an object that represents this connection.

**Syntax**

protected abstract URLConnection openConnection(URL u) throws IOException

**Parameters**

***u***

The URL object specifying the Uniform Resource Locator to which the connection should be established.

**Description**

This method must be overridden in the subclass of the URLStreamHandler class that handles a specific protocol. This method should create and return an instance of a subclass of the URLConnection class that handles the connection stream to the protocol-specific data.

**Imports**

*import java.net.URLStreamHandler;*

**Returns**

This method returns an instance of a protocol-specific implementation of the URLConnection class. This object is used to get access to the actual data the URL points to.

**See Also**

The URL.getContent(URL) method

**Example**

Please refer to the example for the constructor method of this class.

## parseURL(URL, String, int, int)

**ClassName**

URLStreamHandler

**Purpose**

Parses a string specification of a Uniform Resource Locator into the context of an existing URL object.

**Syntax**

protected void parseURL(URL u, String spec, int start, int limit)

**Parameters**

***u***

The URL object to use as the context for parsing the string specification (*spec*).

***spec***

A string object that specifies a Uniform Resource Locator as a text string.

***start***

The start index in *spec* at which to start the parsing.

***limit***

The character position in *spec* at which to stop parsing.

**Description**

The string specification *spec* is parsed as a URL. If it is an absolute URL, then the value of the context URL object *u* is set to this new URL. If the string specification is a relative path specification, then this path is parsed into *u*. The *start* parameter usually points to the character position immediately following the ':' character in a URL, and the limit index is normally the last character in the string or the position of the '#' reference mark. When a URL object is created, a URLStreamHandler object for the protocol specified in the URL is created and this method is invoked on that URLStreamHandler object.This protected method can be invoked only by the other classes in the *java.net* package.

**Imports**

*import java.net.URLStreamHandler;*

**Returns**

None. The result of the parsing is stored in the URL object *u*.

**See Also**

The URL class constructors described in this chapter

**Example**

This method can only be invoked by classes in the *java.net* package and hence no example code illustrating the usage of this method is provided here.

## setURL(URL)

**ClassName**

URLStreamHandler

**Purpose**

Sets the fields of the specified URL object. This is used when a string specification is to be parsed into the context of an existing URL object.

**Syntax**

protected void setURL(URL u, String protocol, String host, int port, String file, String ref)

**Parameters**

***u***

The URL object that is to be modified.

***protocol***

The protocol (http, news, etc) to use for the URL.

***host***

The Internet name of the host machine.

***port***

The port number on the host machine.

***file***

The path name of the file on the host.

***ref***

The name of the reference that indicates a specific offset (to an anchor) into the file.

**Description**

This protected method is invoked by the parse(URL, String, int, int) method of this class to set the individual fields of a URL that is used as a context for parsing a string specification into.

**Imports**

*import java.net.URLStreamHandler;*

**Returns**

None.

**See Also**

The parse method of this class

**Example**

This method can only be invoked by classes in the *java.net* package and hence no example code illustrating the usage of this method is provided here.

## toExternalForm(URL)

**ClassName**

URLStreamHandler

**Purpose**

Represents the specified URL object as a text string.

**Syntax**

protected String toExternalForm(URL u)

**Parameters**

***u***

The URL object that is to be represented as a plain text string.

**Description**

The specified URL object is queried to extract the individual fields (such as protocol, hostname, port, file, and reference tag) and these values are concatenated to form a text string. This text string is then returned to the invoker of this method. This is a protected method and can be accessed only by the other classes in the *java.net* package.

**Imports**

*import java.net.URLStreamHandler;*

**Returns**

This method returns a text string that represents the specified UniformResource Locator.

**See Also**

The toString and toExternalForm methods of the URL class, described in this chapter

## URLStreamHandlerFactory

**Purpose**

Defines the interface that must be implemented by a class that knows how to create an instance of a specific subclass of URLStreamHandler for a specific protocol.

**Syntax**

public interface URLStreamHandlerFactory extends Object

**Description**

A class that implements this interface must know the specific subclass of URLStreamHandler that needs to be created for a protocol. As mentioned earlier in this chapter, the URLStreamHandler class needs to be subclassed to implement a URLStreamHandler for each protocol that is supported by the browser. The implementation details of constructing specific instances of the subclasses of URLStreamHandler are hidden behind this interface. Figure 11-9 shows the inheritance diagram for the URLStreamHandlerFactory interface.

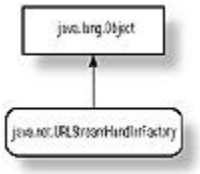


**Figure 11-9** Inheritance diagram for the URLStreamHandlerFactory interface

**PackageName**

*java.net*

**Imports**

*import java.net.URLStreamHandlerFactory;*

**Constructors**

None.

**Parameters**

None.

## createURLStreamHandler(String)

**InterfaceName**

URLStreamHandlerFactory

**Purpose**

Creates an instance of a subclass of the URLStreamHandler class that knows how to create and handle streams for a specified protocol.

**Syntax**

public abstract URLStreamHandler createURLStreamHandler(String protocol)

**Parameters**

***protocol***

The protocol for which an instance of a specific subclass of the URLStreamHandler class (that handles the specified protocol) needs to be created.

**Description**

To implement a protocol handler in Java, you must extend the URLStreamHandler class and implement the specifics of the protocol. One of the data members of the URL class is an instance of a class that implements the URLStreamHandlerFactory interface. This data member is the same for all instances of the URL class (i.e., it is a static member) and can be set using the setURLStreamHandlerFactory method of the URL class. When you construct an URL object, the URL class invokes this method on its factory object to create an instance of the URLStreamHandler subclass that handles the protocol specified in the constructor of the URL object.

**Imports**

*java.net.URLStreamHandlerFactory*

**Returns**

This method returns an instance of a subclass of the URLStreamHandler object that handles the specified protocol (http, nntp, and so on).

**See Also**

The setURLStreamHandlerFactory of the URL class described in this chapter

**Example**

The following Java class implements the URLStreamHandlerFactory interface.

```
// File: CustomURLStreamHandlerFactory.java
//         A sample implementation of a the URLStreamHandlerFactory
import java.net.*;
public class CustomURLStreamHandlerFactory implements URLStreamHandler
Factory {
        public URLStreamHandler createURLStreamHandler(String protocol)
{
                // In this case, all the protocol handlers are
                located relative to the
                // custom/protocol directory. The custom/protocol
                directory resides
                // under the directory hierarchy where the browser
                knows to look for
                // the Java .class files
                // By convention, the URLStreamHandler class is
                always called Handler and
                // it is always created by referring to the
                absolute pathname for the class
                // By referring to the class using its absolute
                pathname specification, the
                // ambiguity of all the URLStreamHandler
                subclasses being called Handler is resolved.
                // For example, the URLStreamHandler class for
                the http protocol would be placed in a
                // file called Handler.java in the subdirectory
                "http" under the custom/protocol
```

```
                    // directory and it would be created by creating
                    an instance of the
                    // custom.protocol.http.Handler class
                    URLStreamHandler handler = null;
                    try {
                              // Construct the absolute pathname of the
                              URLStreamHandler subclass
                              // that implements the specified protocol
                              String absClassName = "custom.protocol."
                              + protocol + ".Handler";

                              // now create an instance of the class
                              and return it
                              handler = (URLStreamHandler)Class.forName
                              (absClassName).newInstance();
                    } catch (Exception e) {
                              // Error handling code should go here
                    }
                    return handler;
          }
}
```

## MalformedURLException

**Purpose**

Signals that the specified Uniform Resource Locator (URL) is invalid.

**Syntax**

public class MalformedURLException extends IOException

**Description**

This exception is used to indicate that the URL specified is not valid. It should be used by applets to indicate that an error occurred because the specified URL was not valid. A URL that specifies an unsupported protocol is one example of a case where this exception is thrown to indicate an error in the specified protocol. Figure 11-10 shows the inheritance diagram for the MalformedURLException class.

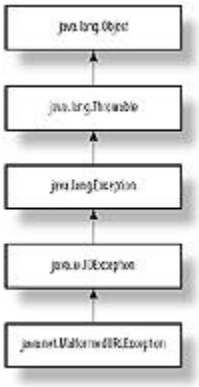

**Figure 11-10** Inheritance diagram for the MalformedURLException class

**PackageName**

*java.net*

**Imports**

*import java.net.MalformedURLException;*

**Constructors**

public MalformedURLException()

public MalformedURLException(String *msg*)

**Parameters**

***msg***

A placeholder for a message that can be used to give additional information to the user about the error that triggered this exception.

**Example**

The following example shows how this exception can be caught in an application.

```
import java.net.*;

public class TestCatchingExceptions {
    public static void main(String args[]) {
        try {
            URL u1 = new URL("http//web.syr.edu/");      // Missing
            colon(:) in the URL specification
        } catch (MalformedURLException e) {
            System.out.println("Caught MalformedURLException !");
        }
    }
}
```

## ProtocolException

**Purpose**

Indicates that an EPROTO error was detected when the application tried to connect to a socket.

**Syntax**

public class ProtocolException extends IOException

**Description**

This exception specifically indicates that a protocol (EPROTO) error was detected when the application tried to connect to a socket. Figure 11-11 shows the inheritance diagram for the ProtocolException class.

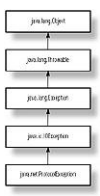

**Figure 11-11** Inheritance diagram for the ProtocolException class

**PackageName**

*java.net*

**Imports**

*import java.net.ProtocolException;*

**Constructors**

public ProtocolException()

public ProtocolException(String *msg*)

**Parameters**

***msg***

This string can be used to give a specific description of the error that caused the ProtocolException to be thrown.

**Example**

The following code sample shows how you use a try/catch statement pair to catch this exception.

```
import java.net.ProtocolException;
        ....
        try {
                // method that throws a ProtocolException
                ...
        } catch (ProtocolException e) {
                // error handling code for this exception
        }
        ....
```

## SocketException

**Purpose**

Indicates that an error occurred during an operation using a socket.

**Syntax**

public class SocketException extends IOException

**Description**

This exception is used to indicate errors that occur while operations are being performed on sockets. Applications may specify additional details of the error that caused this exception to be thrown in the *msg* parameter of the constructor for this exception. Figure 11-12 shows the inheritance diagram for the SocketException class.

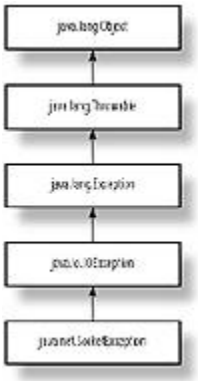

**Figure 11-12** Inheritance diagram for the SocketException class

**PackageName**

*java.net*

**Imports**

*import java.net.SocketException;*

**Constructors**

public SocketException()

public SocketException(String *msg*)

**Parameters**

***msg***

This string can be used to give a specific description of the error that occurred while using the socket.

**Example**

The following code sample shows how you use a try/catch statement pair to catch this exception.

```
import java.net.SocketException;
        .....
        try {
                // method that throws a SocketException
                ...
        } catch (SocketException e) {
                // error handling code for this exception
        }
        ....
```

## UnknownHostException

**Purpose**

Indicates that the address of the host specified by a network client is not valid.

**Syntax**

public class UnknownHostException extends IOException

**Description**

This exception is thrown when the host address (specified by an application that is trying to connect to a server) cannot be resolved as a valid address. Applications that use the socket classes and the classes that deal with Uniform Resource Locators use this class to signal and handle error conditions. Figure 11-13 shows the inheritance diagram for the UnknownHostException class.

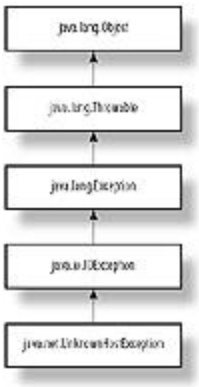

**Figure 11-13** Inheritance diagram for the UnknownHostException class

**PackageName**

*java.net*

**Imports**

*import java.net.UnknownHostException;*

**Constructors**

public UnknownHostException()

public UnknownHostException(String *msg*)

**Parameters**

***msg***

A placeholder for a message that can be used to give additional information to the user about the error that triggered this exception.

**Example**

The following code sample shows how you use a try/catch statement pair to catch this exception.

```
import java.net.UnknownHostException;
        .....
        try {
                // method that throws an UnknownHostException
                ...
        } catch (UnknownHostException e) {
                // error handling code for this exception
        }
        ....
```

## UnknownServiceException

**Purpose**

Signals an error indicating that the requested service is not supported by the client-server protocol.

**Syntax**

public class UnknownServiceException extends IOException

**Description**

This exception is used to indicate that a particular service is not recognized by the protocol being used to effect the network transaction. This is intended primarily for use by developers of protocol handlers. Protocol handlers will extend the

URLConnection class to handle specific protocols and it is in this class that this exception will be used to indicate that a service is not supported by the protocol handler. Figure 11-14 shows the inheritance diagram for the UnknownServiceException class.

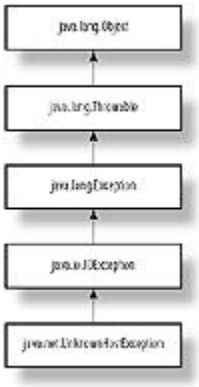

**Figure 11-14** Inheritance diagram for the UnknownServiceException class

**PackageName**

*java.net*

**Imports**

*import java.net.UnknownServiceException;*

**Constructors**

public UnknownServiceException()
public UnknownServiceException(String *msg*)

**Parameters**

***msg***

A placeholder for a message that can be used to give additional information to the user about the error that triggered this exception.

**Example**

The following code sample shows how you use a try/catch statement pair to catch this exception.

```
import java.net.UnknownServiceException;
        .....
        try {
                // method that throws an UnknownServiceException
                ...
        } catch (UnknownServiceException e) {
               // error handling code for this exception
        }
        ....
```

## The URL Class Project

This applet invokes many of the methods in the URL class. By looking at this applet and running it, you will become more familiar with using the URL class. The different types of constructors for URL objects are implemented in this example and the individual fields of a URL are parsed and displayed on the screen. Data contained in the URL object is

read from it and printed in the TextArea component of the URLtest applet. Figure 11-15 shows the URLtest applet in action.

**Figure 11-15** The URLtest applet in action

## Building the Project

This applet divides its display area into two regions. In the first region (represented by class URLPanel), the various parameters of the URL object are displayed to the user. The second region is the control panel that controls which URL's fields are displayed in the URLPanel area. The URLTestControls class implements this simple control panel.

**1.** The class name for the applet is URLtest, so edit a new file named URLtest.java and type the following code into this file. (As always, you must first import the necessary java packages.)

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

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

**2.** Now create the URLtest class which simply extends the java.applet.Applet class. It uses the BorderLayout layout manager to lay out the URLPanel and the URLTestControls objects one below the other. The event handler for this class traps the WINDOW_DESTROY event and exits if the user closes the application. It has a main method so that it can be run as an applet or as a stand-alone Java program.

```
public class URLtest extends Applet {
    public void init() {
                setLayout(new BorderLayout());
                URLPanel up = new URLPanel();
                add("Center", up);
                add("South",new URLTestControls(up));
    }
    public boolean handleEvent(Event e) {
                switch (e.id) {
                case Event.WINDOW_DESTROY:
                        System.exit(0);
                        return true;
                default:
                        return false;
                }
    }
    public static void main(String args[]) {
```

```
                Frame f = new Frame("URLtest");
                URLtest URLTest = new URLtest();
                URLTest.init();
                URLTest.start();

                f.add("Center", URLTest);
                f.resize(500, 500);
                f.show();
    }
}
```

**3.** The URLPanel displays the field parameters of the URL that was selected in the Choice box of the URLTestControls area. This class sets up labels for the fields of the URL and arranges these labels in a two-column format. A TextArea component is used to display the data read from the URL object.

```
class URLPanel extends Panel {
        Label Protocol;
        Label Host;
        Label Port;
        Label Filename;
        Label RefTag;
        Label asString;
        Label ErrorStatus;
        TextArea txtArea;
```

**4.** The constructor for the URLPanel class creates a label component for each field. It also creates a TextArea component that is used to display the data read from the URL object. The descriptive tags and fields are arranged side by side in a two-column format within a separate panel. This panel uses the GridLayout layout manager to lay out the components in two columns. The TextArea component is laid out below the panel containing the labels using the BorderLayout layout manager.

```
 public URLPanel() {
                setLayout(new BorderLayout());

                Panel p1 = new Panel();      // put the fields on
                a separate panel
                p1.setLayout(new GridLayout(0, 2));
                p1.add(new Label("Protocol: "));

                Protocol = new Label();
                p1.add(Protocol);
                p1.add(new Label("Host Name: "));

                Host = new Label();
                p1.add(Host);
                p1.add(new Label("Port Number: "));

                Port = new Label();
                p1.add(Port);
                p1.add(new Label("Filename: "));

                Filename = new Label();
                p1.add(Filename);
                p1.add(new Label("Reference Tag: "));
```

```
                RefTag = new Label();
                p1.add(RefTag);
                p1.add(new Label("Text string: "));

                asString = new Label();
                p1.add(asString);
                p1.add(new Label("Status: "));

                ErrorStatus = new Label();
                ErrorStatus.setBackground(Color.red);
                p1.add(ErrorStatus);

                p1.add(new Label("Contents of the URL"));
                add("Center", p1);

                txtArea = new TextArea("", 15, 80);
                txtArea.setEditable(false);
                // add the TextArea below the panel containing
                // the fields
               add("South", txtArea);
}
```

**5.** The next method is invoked to parse a URL string specification and display the individual field values. A URL object is created using the specified string and the individual fields of the URL are extracted using the methods of the URL class. The contents of the URL object are read and these contents are displayed in the TextArea component. An error message is printed in the TextArea if an exception was caught while reading the contents of the URL. If the specified URL string is not a valid URL then the MalformedURLException is trapped and an error message is displayed on the ErrorStatus Label component of this panel.

```
public void showParams(String urlString) {
               try {
                        URL url1 = new URL(urlString);
                        Protocol.setText(url1.getProtocol());
                        Host.setText(url1.getHost());
                        int port = url1.getPort();
                        if (port != -1)

Port.setText(String.valueOf(port));
                        else
                                 Port.setText("Default port");
                        Filename.setText(url1.getFile());
                        RefTag.setText(url1.getRef());
                        asString.setText(url1.toString());
                        ErrorStatus.setText("Status okay");
                        try {
                                 txtArea.setText("");    // clear
                                 the TextArea
                                 // open an input stream to the
                                 URL object
                                 DataInputStream d = new
DataInput
                                 Stream(url1.openStream());
                                 String line;
                                 while ((line = d.readLine()) !=
                                 null)    // read data and add it
```

```
                                txtArea.appendText(line + "\n");
                                // to the TextArea
                                d.close();     // close the
input
                                stream
                        } catch (IOException ie) {    //
                        Error reading data from the URL object
                                txtArea.setText("Could not
                                read data from the URL!");
                        }
    } catch(MalformedURLException mue) {
                        ErrorStatus.setText("Caught Exception");
    }
}
```

**6.** This event handler exits if the user quits the application.

```
  public boolean handleEvent(Event e) {
            switch (e.id) {
                    case Event.WINDOW_DESTROY:
                            System.exit(0);
                            return true;
                    default:
                            return false;
            }
  }
}
```

**7.** All you have left to do is put up the main control panel for this applet. The URLTestControls class is the control panel. It displays a Choice component with different URLs in it that the user can select. When a URL is selected, the fields of the URL are displayed in the URLPanel area. This class needs to notify the URLPanel whenever a selection is made, so it keeps a reference to the URLPanel object. The constructor utilizes the different forms of constructors for the URL class and adds the string representation of these URLs to the Choice box. An invalid URL specification is also added to the list of choices. This will enable you to see how exceptions are caught.

```
class URLTestControls extends Panel {
    URLPanel target;

    public URLTestControls(URLPanel target) {
                this.target = target;
                setLayout(new FlowLayout());
                Choice urls = new Choice();
                try {
                        // you can experiment by substituting
other
                        URLs in place
                        // of the ones listed here
                        URL url1 = new URL("file", "www.syr.edu",
                        8080, "/docs/intro.html");
                        URL url2 = new URL("http", "www.syr.edu",
                        "/~username/report.ps");
                        URL url3 = new URL("http://cosmos/
                        ~asriniva/test.html");
                        URL url4 = new URL(url3, "http://www.foo.
                        org/applets.html#head4");
```

```
                        URL url5 = new URL(url3,
"docs/hello.gif");
                        urls.addItem("http://www.syr.edu/");
                        urls.addItem(url1.toString());
                        urls.addItem(url2.toString());
                        urls.addItem(url3.toString());
                        urls.addItem(url4.toString());
                        urls.addItem(url5.toString());

urls.addItem("invalid://URLspecification");
            } catch (MalformedURLException mue) {
                        System.out.println("Caught MalformedURL
                        Exception");
                        System.exit(0);
            }
            add(urls);
            this.target.showParams("http://www.syr.edu/");
}
```

**8.** When a selection is made from the list of choices, the event handler notifies the URLPanel to update the URL field parameters, as seen in the following example.

```
    public boolean action(Event e, Object arg) {
                if (e.target instanceof Choice) {
                        String choice = (String)arg;
                        target.showParams(choice);
                        return true;
                } else if (e.id == Event.WINDOW_DESTROY) {
                        System.exit(0);
                        return true;
                }
                return false;
    }
}
```

**9.** That's it, you have finished creating the applet. Now save this file and compile it by typing `javac URLtest.java`. Then run the application by typing `java URLtest`.

## Building the Project

This applet divides its display area into two regions. In the first region (represented by class URLPanel), the various parameters of the URL object are displayed to the user. The second region is the control panel that controls which URL's fields are displayed in the URLPanel area. The URLTestControls class implements this simple control panel.

**1.** The class name for the applet is URLtest, so edit a new file named URLtest.java and type the following code into this file. (As always, you must first import the necessary java packages.)

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

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

**2.** Now create the URLtest class which simply extends the java.applet.Applet class. It uses the BorderLayout layout manager to lay out the URLPanel and the URLTestControls objects one below the other. The event handler for this class traps the WINDOW_DESTROY event and exits if the user closes the application. It has a main method so that it can be run as an applet or as a stand-alone Java program.

```
public class URLtest extends Applet {
    public void init() {
                setLayout(new BorderLayout());
                URLPanel up = new URLPanel();
                add("Center", up);
                add("South",new URLTestControls(up));
    }
    public boolean handleEvent(Event e) {
                switch (e.id) {
                case Event.WINDOW_DESTROY:
                        System.exit(0);
                        return true;
                default:
                        return false;
                }
    }
    public static void main(String args[]) {
                Frame f = new Frame("URLtest");
                URLtest URLTest = new URLtest();
                URLTest.init();
                URLTest.start();

                f.add("Center", URLTest);
                f.resize(500, 500);
                f.show();
    }
}
```

**3.** The URLPanel displays the field parameters of the URL that was selected in the Choice box of the URLTestControls area. This class sets up labels for the fields of the URL and arranges these labels in a two-column format. A TextArea component is used to display the data read from the URL object.

```
class URLPanel extends Panel {
        Label Protocol;
        Label Host;
        Label Port;
        Label Filename;
        Label RefTag;
        Label asString;
        Label ErrorStatus;
        TextArea txtArea;
```

**4.** The constructor for the URLPanel class creates a label component for each field. It also creates a TextArea component that is used to display the data read from the URL object. The descriptive tags and fields are arranged side by side in a two-column format within a separate panel. This panel uses the GridLayout layout manager to lay out the components in two columns. The TextArea component is

laid out below the panel containing the labels using the BorderLayout layout manager.

```
 public URLPanel() {
                setLayout(new BorderLayout());

                Panel p1 = new Panel();      // put the fields on
                a separate panel
                p1.setLayout(new GridLayout(0, 2));
                p1.add(new Label("Protocol: "));

                Protocol = new Label();
                p1.add(Protocol);
                p1.add(new Label("Host Name: "));

                Host = new Label();
                p1.add(Host);
                p1.add(new Label("Port Number: "));

                Port = new Label();
                p1.add(Port);
                p1.add(new Label("Filename: "));

                Filename = new Label();
                p1.add(Filename);
                p1.add(new Label("Reference Tag: "));

                RefTag = new Label();
                p1.add(RefTag);
                p1.add(new Label("Text string: "));

                asString = new Label();
                p1.add(asString);
                p1.add(new Label("Status: "));

                ErrorStatus = new Label();
                ErrorStatus.setBackground(Color.red);
                p1.add(ErrorStatus);

                p1.add(new Label("Contents of the URL"));
                add("Center", p1);

                txtArea = new TextArea("", 15, 80);
                txtArea.setEditable(false);
                // add the TextArea below the panel containing
                // the fields
               add("South", txtArea);
}
```

**5.** The next method is invoked to parse a URL string specification and display the individual field values. A URL object is created using the specified string and the individual fields of the URL are extracted using the methods of the URL class. The contents of the URL object are read and these contents are displayed in the TextArea component. An error message is printed in the TextArea if an exception was caught while reading the contents of the URL. If the specified URL string is not a valid URL then the MalformedURLException is trapped and an error message is displayed on the ErrorStatus Label component of this panel.

```
public void showParams(String urlString) {
               try {
                        URL url1 = new URL(urlString);
                        Protocol.setText(url1.getProtocol());
                        Host.setText(url1.getHost());
                        int port = url1.getPort();
                        if (port != -1)

Port.setText(String.valueOf(port));
                        else
                                 Port.setText("Default port");
                        Filename.setText(url1.getFile());
                        RefTag.setText(url1.getRef());
                        asString.setText(url1.toString());
                        ErrorStatus.setText("Status okay");
                        try {
                                 txtArea.setText("");    // clear
                                 the TextArea
                                 // open an input stream to the
                                 URL object
                                 DataInputStream d = new
DataInput
                                 Stream(url1.openStream());
                                 String line;
                                 while ((line = d.readLine()) !=
                                 null)    // read data and add it
                                 txtArea.appendText(line + "\n");
                                 // to the TextArea
                                 d.close();     // close the
input
                                 stream
                        } catch (IOException ie) {    //
                        Error reading data from the URL object
                                     txtArea.setText("Could not
                                     read data from the URL!");
                        }
    } catch(MalformedURLException mue) {
                        ErrorStatus.setText("Caught Exception");
    }
}
```

**6.** This event handler exits if the user quits the application.

```
  public boolean handleEvent(Event e) {
             switch (e.id) {
                      case Event.WINDOW_DESTROY:
                              System.exit(0);
                              return true;
                      default:
                              return false;
             }
  }
}
```

**7.** All you have left to do is put up the main control panel for this applet. The URLTestControls class is the control panel. It displays a Choice component with different URLs in it that the user can select. When a URL is selected, the fields of the URL are displayed in the URLPanel area. This class needs to notify the

URLPanel whenever a selection is made, so it keeps a reference to the URLPanel object. The constructor utilizes the different forms of constructors for the URL class and adds the string representation of these URLs to the Choice box. An invalid URL specification is also added to the list of choices. This will enable you to see how exceptions are caught.

```
class URLTestControls extends Panel {
    URLPanel target;

    public URLTestControls(URLPanel target) {
                this.target = target;
                setLayout(new FlowLayout());
                Choice urls = new Choice();
                try {
                        // you can experiment by substituting
other
                        URLs in place
                        // of the ones listed here
                        URL url1 = new URL("file", "www.syr.edu",
                        8080, "/docs/intro.html");
                        URL url2 = new URL("http", "www.syr.edu",
                        "/~username/report.ps");
                        URL url3 = new URL("http://cosmos/
                        ~asriniva/test.html");
                        URL url4 = new URL(url3, "http://www.foo.
                        org/applets.html#head4");
                        URL url5 = new URL(url3,
"docs/hello.gif");
                        urls.addItem("http://www.syr.edu/");
                        urls.addItem(url1.toString());
                        urls.addItem(url2.toString());
                        urls.addItem(url3.toString());
                        urls.addItem(url4.toString());
                        urls.addItem(url5.toString());

urls.addItem("invalid://URLspecification");
            } catch (MalformedURLException mue) {
                        System.out.println("Caught MalformedURL
                        Exception");
                        System.exit(0);
            }
            add(urls);
            this.target.showParams("http://www.syr.edu/");
}
```

**8.** When a selection is made from the list of choices, the event handler notifies the URLPanel to update the URL field parameters, as seen in the following example.

```
    public boolean action(Event e, Object arg) {
                if (e.target instanceof Choice) {
                        String choice = (String)arg;
                        target.showParams(choice);
                        return true;
                } else if (e.id == Event.WINDOW_DESTROY) {
                        System.exit(0);
                        return true;
                }
```

```
                return false;
        }
}
```

**9.** That's it, you have finished creating the applet. Now save this file and compile it by typing `javac URLtest.java`. Then run the application by typing `java URLtest`.

### How It Works

This project illustrates some of the methods of the URL class. These methods are applied to different URL strings and the results of these methods are displayed. This project also shows how you can use the URL class to read data from a remote URL object. Java programs written for Internet applications will find the URL class invaluable. Classes such as the ContentHandler class and the URLStreamHandler class can easily be extended to support custom data formats and protocols.These custom formats can then be viewed using the class extensions. The classes and interfaces described in this chapter provide functionality for accessing data on the Internet, and for sending to and receiving information from the World Wide Web.

# *Part V*
# *Java Utilities*

# Chapter 12
# Data Structures And Random Number Generation

Java's utility classes provide a wide range of functionality for your applications. This chapter explains the utility classes for implementing data structures and random number generation. These classes form a part of the java.util package in the Java APIs. We will look at the data structure classes Dictionary, Hashtable, Vector, and Stack as well as the Properties class for maintaining the properties of objects. The Enumeration interface, which provides the methods necessary to enumerate a given set of elements from the vector or hash table is also covered, along with two additional classes: EmptyStackException and NoSuchElementException, which define the exceptions related to these data structures. The project developed at the end of this chapter is an Appointment Organizer application. It's a Java application that provides a user-interface to enter appointment details. Details about an appointment including the date and time are entered using text fields and a text area. Specifying the date and time provides a facility for retrieving the details. The classes discussed in this chapter will be used in this application.

## Dictionary and Hashtable

Dictionary is an abstract class that maps keys to values in a table. It forms the super class of Hashtable. You should use hash tables when you want to store and retrieve elements efficiently. You can associate a key with each element you store, so that you can use the key to retrieve the element quickly. A function is used to map the key to a particular slot in the hash table. You should also note that there can be only one element for a given key in the hash table at a given time. As an example, consider the elements Arizona, California, New York, and Oregon. If you decide that the key for these elements is their first character, then mapping them to a hash table will result in the table shown in Figure 12-1(a). If a new element, North Carolina is listed after New York, then two elements will have the same key (N) resulting in a collision. In the Hashtable implementation of Java, when collisions occur, the new element replaces the old element. In our example, the element North Carolina will replace the element New York, resulting in the hash table shown in Figure 12-1(b).

**Figure 12-1** Two hashtables (a) States with one string element (b) North Carolina replaces New York as both have the same key (N)

Note that we have implicitly assumed in our example that the size of the hash table is 26, mapping to the 26 characters in the English alphabet. This is called the capacity of the hash table. You can specify the capacity of the hash table when you create an instance of Hashtable. You can also specify the load factor. The load factor is the ratio of the number of elements in the hash table to the size of the hash table. In our example, the load factor is 4/26. Note that the load factor has a value between 0.0 and 1.0.

In Java, the Hashtable is a subclass of the Dictionary class. The Dictionary class encapsulates the characteristics of mapping keys to values. The Hashtable object expands automatically when it gets full. It will maintain the capacity such that the load factor of the table at any instance is at least the specified load factor. The elements can be any of the Objects in Java. So you can also have a list of strings as an element. Considering our example hash table of states, we can specify that elements with the same first character (key) form a growable list. So if we add North Carolina to the hash table, instead of replacing New York it will be added to the list of string with 'N' as the first character. Figure 12-2 shows the resulting table.


**Figure 12-2** A hash table with the element being a list

## Vector and Stack

The Vector class in Java represents an extensible array. You will often come across situations where you do not know the number of items in an array in advance. In such situations, you can use Vector. Imagine you were having a party to celebrate a promotion. You might arrange ten chairs in the living room to start. When an eleventh person arrives, you could fetch another chair, and continue adding one chair for every new guest, or you could bring three more chairs every time you run out of seats. (See Figure 12-3.) A Vector is like adding three chairs at a time, but the increased space is a memory unit, i.e., on demand for more space, new space gets allocated automatically. As more elements are stored in a Vector, the vector's size increases. You can specify it to increase by increments of size defined by capacityIncrement, a variable defined in the Vector class.

**Figure 12-3** Providing seats for incoming guests (a) An eleventh guest needs a chair (b) Increase the number of chairs by five

A common data structure used in many applications is a stack. It is an ordered vector (Stack subclasses Vector) in which all insertions and deletions are made at one end, called the top. Consider four elements A, B, C, and D inserted in a stack in that order. Stack is a Last-In-First-Out structure, so A would be the bottommost element and D would be the topmost element, as shown in Figure 12-4. You can access B only after popping D and C off the stack. Stacks are useful when you want to access the most recently created items and only after using them will you get to use earlier items. Compilers typically use stacks to implement computations effectively. As a matter of fact, the Java interpreter is a stack-based interpreter compared to other register-based interpreters.

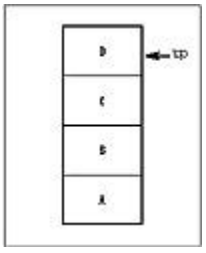


**Figure 12-4** A stack of elements

## Random Numbers

Random numbers are important in algorithms where you want to provide a variation in behavior or when you don't have a fixed option to consider. Assume you have five options to select from, namely: A, B, C, D, E. If you don't have any particular preference, you'll select one of them arbitrarily. Simulating this behavior in a computer program is achieved using random numbers. Pure random numbers are very difficult to generate, but a set of pseudo-random numbers can be generated by many available algorithms. Suppose you divide the interval between 0 and 1 into five equals units. You generate a

random number and depending on which interval it occupies, one of the 5 elements is selected, as shown in Figure 12-5. For more accuracy you might want to generate double precision random numbers. Or you might be interested in integer random numbers. In Java, the Random class provides the methods to generate pseudo-random numbers. You can provide a seed to the generator so that it generates a repeatable set of pseudo-random numbers. This is necessary for users who repeat experiments and study the results at similar circumstances.

**Figure 12-5** Random selection among 5 elements

Enumeration is a Java interface that provides a set of methods to enumerate or a method to count a set of objects. In the case of an array, you can access elements in the array by using the index numbers. But in the case of Vector or Stack, you don't have fixed indices. To index into these data structures, the Enumeration class is useful. It helps you to enumerate or count (and effectively index) the elements. The NoSuchElementException and StackEmptyException classes are defined to throw runtime exceptions while handling invalid elements in the data structure classes in Java.

## Data Structure, Properties, and Random Class Summaries

Table 12-1 summarizes the classes and interface that help implement efficient algorithms using predefined data structures and utility class for random number generation.

**Table 12-1** Class and interface summary for Data structure, Properties and Random classes

| Class/Interface Name | Description |
|---|---|
| Dictionary | Maps keys to values and is the parent of Hashtable. |
| Hashtable | Encapsulates the features of a hash table to store and retrieve values efficiently. |
| Properties | Contains the persistent properties of the associated object. |
| Vector | Represents an extensible array, designed for space optimization. |
| Stack | Encapsulates the Last-In-First-Out(LIFO) stack of objects. |
| Enumeration | Interface that specifies methods to enumerate a set of objects. |
| Random | Generates pseudo-random numbers. |
| EmptyStackException | A Runtime exception that gets thrown when trying to pop out an object from an empty stack. |

| | |
|---|---|
| NoSuchElementException | A Runtime exception signaling an empty enumeration. |

## Dictionary

**Purpose**

An abstract class that maps keys to values and is the parent of Hashtable.

**Syntax**

public abstract class Dictionary extends Object

**Description**

Dictionary is an abstract class and the superclass of Hashtable. It maps keys to values. Any object can be used in a Dictionary as a key and/or a value. The Dictionary data structure improves search time for an element by use of its key. Operations on elements in a Dictonary can be carried out using its key. Figure 12-6 illustrates the inheritance relationship of class Dictionary.

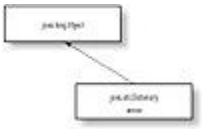

**Figure 12-6** Class diagram of Dictionary class

**PackageName**

*java.util*

**Imports**

*import java.util.Dictionary;*

**Constructors**

None.

**Parameters**

None.

**Example**

The class inventory applet, below, represents a basic inventory maintenance program. It uses a hash table for storing entries. Hashtable is a subclass of the abstract Dictionary class. The program for the inventory class is given in Listing 12-1.

**Listing 12-1** inventory.java: Program illustrating use of hash table for inventory control

```
import java.util.*;

class inventory {

  Hashtable ht;
  Hashtable backup;
```

```
 public inventory() {

       ht = new Hashtable(5);
 }

protected void populate() {
       ht.put("dairy", "Milk");
       ht.put("mag", "Time");
       ht.put("prod", "Tomato");
       ht.put("can", "Soup");
       ht.put("elec", "Bulb");
       backup = (Hashtable)ht.clone();
}

protected void addCheese() {
       ht.put("dairy", "Cheese");
}

protected void addCereal() {
       ht.put("food", "Cereal");
}

protected void listAll() {
       if (ht.isEmpty()){
              System.out.println(" Hash Table is empty ");
             return;
       }

       System.out.println(" Hash table contains the following
       keys/items");
       for (Enumeration e = ht.elements(), k = ht.keys();
       e.hasMoreElements();){
               System.out.println(" " + k.nextElement()
       + " " + e.nextElement());
       }
}

protected void removeSoup() {
       if (ht.remove("can") ==null)
               System.out.println(" removal of can item
               unsuccessful");
       System.out.println(" Size of hash table is now " +
       ht.size());

}

protected void reset() {
       ht.clear();
       populate();
}

private void table() {
       System.out.println(" The table's string form is " +
       ht.toString());
}
protected void findChanges() {
```

```
        System.out.println(" Following keys/elements are newly
        added");
        for (Enumeration e = ht.elements(), k = ht.keys();
        e.hasMoreElements();){
                Object elem = e.nextElement();
                Object key = k.nextElement();
                if (!backup.contains(elem))
                        System.out.println(key + " " +
                        elem);
                if (!backup.containsKey(key))
                        System.out.println(" Also there is no
                        element in " + key + " section ");
              }

 }

 protected void getItem(String key) {
        System.out.println(" Under the section " + key);
        Object item = ht.get(key);
        if (item == null)
                System.out.println(" no item found");
        else
                System.out.println(" The item " + item +
                " is available");
 }
 public static void main(String args[]) {
        inventory inv = new inventory();
        inv.populate();
        inv.listAll();
        inv.addCheese();
     System.out.println(" After adding cheese … \n");
        inv.listAll();
        inv.addCereal();
     System.out.println(" After adding cereal … \n");
        inv.listAll();
        inv.removeSoup();
     System.out.println(" After removing soup … \n");
        inv.listAll();
  // System.out.println(" After resetting the hash table … \n");
  //    inv.reset();
  //    inv.listAll();
        inv.findChanges();
        inv.getItem("dairy");
     inv.table();
  }
}
```

## elements()

**ClassName**

Dictionary

**Purpose.**

Provides an enumeration of the elements.

**Syntax**

public abstract Enumeration elements().

**Parameters**

None.

**Description**

The method returns an enumeration of the elements in Dictionary. This is an abstract method and should be overridden in its subclass. The enumeration that is returned can be used to fetch the elements sequentially.

**Imports**

*java.utils.Dictionary*

**Returns**

An Enumeration of elements. Return type is Enumeration.

**See Also**

The keys method in Dictionary class; Enumeration interface

**Example**

Refer to the method listAll() in class inventory in Listing 12-1. It gets all the elements by using this method on a Hashtable object.

## get(Object)

**ClassName**

Dictionary

**Purpose**

Obtains the Object associated with the specified key in the Dictionary.

**Syntax**

public abstract Object get(Object key)

**Parameters**

***key***

The key, in the Dictionary, of the object to be fetched.

**Description**

The method returns the Object associated with the key specified. The element for the key is returned if the key is defined in the hash table. If the key is not defined, this method returns null. This is an abstract method and should be overridden in its subclass. The returned object is typecast to a class, which is expected to be returned in the given context.

**Imports**

*import java.utils.Dictionary;*

**Returns**

The element associated with the key. Return type is Object (note that any class is a subclass of Object).

**See Also**

The put method in the Dictionary class

**Example**

Refer to the getItem method in the inventory class in Listing 12-1. It uses this get method to get the element with the specified key.

## isEmpty()

**ClassName**

Dictionary

**Purpose**

Boolean value indicating if the Dictionary contains any elements.

**Syntax**

public abstract boolean isEmpty()

**Parameters**

None.

**Description**

This method returns true if the Dictionary contains no elements. If the Dictionary contains even one element, this method returns false. This is an abstract method and should be overridden in its subclass.

**Imports**

*import java.utils.Dictionary;*

**Returns**

True if empty and false if the Dictionary contains one or more elements.

**See Also**

The elements method in Dictionary class

**Example**

Refer to the listAll()method in the inventory class in Listing 12-1. It checks if the hash table is empty using this method.

## keys()

**ClassName**

Dictionary

**Purpose**

Provides an enumeration of the Dictionary's keys.

**Syntax**

public abstract Enumeration keys()

**Parameters**

None.

**Description**

This method returns an enumeration of the keys in Dictionary. This is an abstract method and should be overridden in its subclass. The enumeration that is returned can be used to list the keys sequentially.

**Imports**

*import java.utils.Dictionary;*

**Returns**

An Enumeration of the Dictionary's keys. Return type is Enumeration.

**See Also**

The elements method in the Dictionary class; the Enumeration interface

**Example**

Refer to the listAll()method in the inventory class in Listing 12-1. It lists all the elements with the keys by using this method on a Hashtable object.

## put(Object, Object)

**ClassName**

Dictionary

**Purpose**

Puts the specified Object into the Dictionary using the specified key.

**Syntax**

public abstract Object put(Object key, Object value)

**Parameters**

***key***

The key, in the Dictionary, of the object to be included.

***value***

The element to be put into the Dictionary.

**Description**

This method puts the specified Object with the given value into the Dictionary using the specified key. After inserting the new value, the old value is removed from the table and returned. The old value corresponding to the key is returned if it existed. If not, this method returns null. This is an abstract method and should be overridden in its subclass.

**Imports**

*import java.utils.Dictionary;*

**Returns**

The element associated with the key. Return type is Object (note that any class is a subclass of Object).

**See Also**

The get method in the Dictionary class

**Example**

Refer to the populate method in the inventory class in Listing 12-1. It populates elements by using this method on a Hashtable object.

## remove(Object)

**ClassName**

Dictionary

**Purpose**

Removes the specified key in the Dictionary.

**Syntax**

public abstract Object remove(Object key)

**Parameters**

***key***

The key of the object to be removed from the Dictionary.

**Description**

This method removes the specified key from the Dictionary. If the specified key is not present in the Dictionary, this method does nothing. This is an abstract method and should be overridden in its subclass. This method returns the value associated with the specified key to be removed. If the key was not found, this method returns null.

**Imports**

*import java.utils.Dictionary;*

**Returns**

The value of the key to be removed or null, if the key is not present in the Dictionary. Return type is Object.

**See Also**

The put and get methods in the Dictionary class

**Example**

Refer to the removeSoup method in the inventory class in Listing 12-1. It removes the Soup item by removing its key "can" by using this method on a Hashtable object.

## size()

**ClassName**

Dictionary

**Purpose**

Obtains the number of elements contained within the Dictionary.

**Syntax**

public abstract int size()

**Parameters**

None.

**Description**

This method returns the number of elements in the Dictionary. This is an abstract method and should be overridden in its subclass.

**Imports**

*import java.utils.Dictionary;*

**Returns**

The number of elements in the Dictionary. Return type is int.

**See Also**

The isEmpty method in the Dictionary class.

**Example**

Refer to the removeSoup method in the inventory class in Listing 12-1. The size of the hash table is obtained using this method.

## Hashtable

**Purpose**

Encapsulates the features of a hash table to store and retrieve values efficiently and maps keys to values

**Syntax**

public class Hashtable extends Dictionary implements Cloneable

**Description**

Hashtable subclasses the Dictionary class. It maps keys to values. Any object can be used as a key and/or value in a hash table. The object that is used as a key must implement the hashCode() and equals() methods to successfully store and retrieve objects from the hash table. The class defines the method clone() as it implements the Cloneable interface. When you create an instance of the hash table, you can specify the initial capacity of the hash table and its desired load factor. The load factor of the hash table is the ratio of the number of elements present in the table to the size of the table. This load factor is used to determine the size of the increment when the hash table's capacity has to be increased. Figure 12-7 illustrates the inheritance relationship of the Hashtable class.

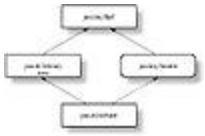

**Figure 12-7** Class diagram of Hashtable class

**PackageName**

*java.util*

**Imports**

import java.util.Hashtable;

**Constructors**

public Hashtable()
public Hashtable(int *init_cap*)
public Hashtable(int *init_cap*, float *load_factor*)

**Parameters**

***init_cap***

Initial capacity of the hash table.

***load_factor***

Load factor of the hash table contains the ratio of the number of elements, in the hash table, to the capacity of the table.

**Example**

The inventory class, in Listing 12-1, contains a member of type Hashtable.

## clear()

**ClassName**

Hashtable

**Purpose**

Clears the hash table so that there are no elements in it.

**Syntax**

public synchronized void clear()

**Parameters**

None.

**Description**

The method removes all the elements in the hash table and clears it.

**Imports**

*import java.utils.Hashtable;*

**Returns**

None.

**Example**

Refer to the reset()method in the inventory class in Listing 12-1. It uses this clear method to clear the hash table.

## clone()

**ClassName**

Hashtable

**Purpose**

Creates and returns a clone of the hash table.

**Syntax**

public synchronized Object clone()

**Parameters**

None.

**Description**

This method creates a clone of the hash table and returns a handle to it. A shallow copy of the hash table is made with its characteristics. None. of the elements or keys of the hash table are cloned in this operation. This cloning operation is relatively expensive. This method overrides the clone method of class Object. Typecast the object returned to Hashtable for further operations. This method is defined as the Hashtable class and implements the Cloneable interface.

**Imports**

*import java.utils.Hashtable;*

**Returns**

A handle to the clone hash table. Return type is Object.

**Example**

Refer to Listing 12-1. In the populate method, this clone method is used to create a backup of the original hash table and is kept for future comparisons.

## contains(Object)

**ClassName**

Hashtable

**Purpose**

Checks if the specified object is an element in the Hashtable.

**Syntax**

public synchronized boolean contains(Object value)

**Parameters**

***value***

The value associated with the element we are searching.

**Description**

This method checks to see if the specified element is in the Hashtable. It returns true if the object is in the hash table, false if it is not. If the specified value is null, this method throws a NullPointerException. Note that searching for a specified object in a hash table is more expensive than searching for a key in the hash table.

**Imports**

*import java.utils.Hashtable;*

**Returns**

If the object is in the hash table, the method returns true. If the object is not in the hash table, the method returns false. Return type is boolean.

**See Also**

The containsKey method in the Hashtable class

**Example**

Refer to the findChanges method in the inventory class in Listing 12-1. It uses the contains method to find out the newly added items.

## containsKey(Object)

**ClassName**

Hashtable

**Purpose**

Checks if an element exists in the hash table associated with the specified key.

**Syntax**

public synchronized boolean containsKey(Object key)

**Parameters**

***key***

The key of the element for which we are searching.

**Description**

This method checks to see if an object associated with the specified key is in the Hashtable. It returns true if the object is in the hash table, false if it is not. If the specified key is null, this method throws a NullPointerException. Note that searching for an object using its key in a hash table is less expensive than searching for the object itself in the hash table.

**Imports**

*import java.utils.Hashtable;*

**Returns**

If the object associated with the specified key is in the hash table, the method returns true. If the object is not in the hash table, the method returns false. Return type is boolean.

**See Also**

The contains method in the Hashtable class

**Example**

Refer to the findChanges method in the inventory class in Listing 12-1. It uses the containsKey method to find the sections with no item.

## elements()

**ClassName**

Hashtable

**Purpose**

An enumeration of the elements is returned.

**Syntax**

public synchronized Enumeration elements()

**Parameters**

None.

**Description**

This method returns an enumeration of the elements in Hashtable. This overrides the elements method in the Dictionary class. The enumeration that is returned can be used to fetch the elements sequentially.

**Imports**

*import java.utils.Hashtable;*

**Returns**

An Enumeration of elements. Return type is Enumeration.

**See Also**

The keys method in the Hashtable class; the Enumeration interface

**Example**

Refer to the listAll() method in the inventory class in Listing 12-1. It gets all the elements by using this method on the Hashtable object.

## get(Object)

**ClassName**

Hashtable

**Purpose**

Returns the Object associated with the specified key in the Hashtable.

**Syntax**

public synchronized Object get(Object key)

**Parameters**

***key***

The key, in the Hashtable, of the object to be fetched.

**Description**

This method returns the Object associated with the specified key. The element for the key is returned if the key is defined in the hash table. If the key is not defined, this method returns null. This overrides the get method in the Dictionary class.

**Imports**

*import java.utils.Hashtable;*

**Returns**

The element associated with the key. Return type is Object.

**See Also**

The put method in the Hashtable class

**Example**

Refer to the getItem method in the inventory class in Listing 12-1. It uses the get method for getting the element with the specified key.

## isEmpty()

**ClassName**
Hashtable
**Purpose**
Boolean value indicating whether or not the Hashtable contains any elements.
**Syntax**
public boolean isEmpty()
**Parameters**
None.
**Description**
This method returns true if the Hashtable contains no elements. If the Hashtable contains even one element, this method returns false. This method overrides the isEmpty method in class Dictionary.
**Imports**
*import java.utils.Hashtable;*
**Returns**
True if empty and false if the Hashtable contains one or more elements.
**See Also**
The elements method in the Hashtable class
**Example**
Refer to the listAll()method in the inventory class in Listing 12-1. It checks if the hash table is empty using this method.

## keys()

**ClassName**
Hashtable
**Purpose**
Obtains an enumeration of the Hashtable's keys.
**Syntax**
public synchronized Enumeration keys()
**Parameters**
None.
**Description**
This method returns an enumeration of the keys in Hashtable. This method overrides the keys method in the Dictionary class. The enumeration that is returned can be used to list the keys sequentially.
**Imports**
*import java.utils.Hashtable;*
**Returns**
An Enumeration of the Hashtable's keys. Return type is Enumeration.
**See Also**
The elements method in the Hashtable class; the Enumeration interface
**Example**

Refer to the listAll()method in the inventory class in Listing 12-1. It lists all the elements with the keys by using this method on a Hashtable object.

## put(Object, Object)

**ClassName**

Hashtable

**Purpose**

Puts the specified Object into the Hashtable using the specified key.

**Syntax**

public synchronized Object put(Object key, Object value)

**Parameters**

***key***

The key, in the Hashtable, of the object to be put.

***value***

The element to be put into the Hashtable.

**Description**

This method puts the specified Object of given value into the Hashtable using the specified key. After inserting the new value, the old value is removed and returned by this method if it existed. If not, this method returns null. The key and the value cannot both be null. The element that is put into the hash table may be retrieved using the get method in the Hashtable class, by specifying the same key. This method overrides the put method in the Dictionary class.

**Imports**

*import java.utils.Hashtable;*

**Returns**

The element associated with the key. Return type is Object.

**See Also**

The get method in Hashtable class

**Example**

Refer to the populate method in the inventory class in Listing 12-1. It populates elements by using this method on a Hashtable object.

## rehash()

**ClassName**

Hashtable

**Purpose**

The content of the hash table is rehashed into a bigger hash table.

**Syntax**

protected void rehash()

**Parameters**

None.

**Description**

This method rehashes the contents of the hash table into a bigger table. This method is automatically invoked when the size of the Hashtable exceeds the threshold as more elements are added to the table. This is a protected method and hence can be used only by the methods within the *java.util* package.

**Imports**

*import java.utils.Hashtable;*

**Returns**

None.

**Example**

This method can be used only in a class that is part of the *java.util* package, so it should have *package java.util* as the first statement in the file. Then it can be used in the class on a hash table object.

## remove(Object)

**ClassName**

Hashtable

**Purpose**

Removes the specified key in the Hashtable.

**Syntax**

public synchronized Object remove(Object key)

**Parameters**

***key***

The key of the object to be removed from the Hashtable.

**Description**

This method removes the specified key from the Hashtable. If the specified key is not present in the Hashtable, there is no effect. This method overrides the remove method in the Dictionary class. This method returns the value associated with the specified key to be removed. If the key is not found, this method returns null.

**Imports**

*import java.utils.Hashtable;*

**Returns**

The value of the key to be removed or null if the key is not present in the Hashtable. The return type is Object.

**See Also**

The put and get methods in class Hashtable

**Example**

Refer to the removeSoup method in the inventory class in Listing 12-1. It removes the Soup item by removing its key “can” by using this method on a Hashtable object.

## size()

**ClassName**

Hashtable

**Purpose**

Obtains the number of elements contained in the Hashtable.

**Syntax**

public int size()

**Parameters**

None.

**Description**

This method returns the number of elements in the Hashtable. This method overrides the size method in the Dictionary class.

**Imports**

*import java.utils.Hashtable;*

**Returns**

The number of elements in the Hashtable. Return type is int.

**See Also**

The isEmpty method in the Hashtable class

**Example**

Refer to the removeSoup method in the inventory class in Listing 12-1. The size of the hash table is obtained using this method.

## toString()

**ClassName**

Hashtable

**Purpose**

Converts the instance of Hashtable to its string form.

**Syntax**

public synchronized String toString()

**Parameters**

None.

**Description**

This method converts the target Hashtable object to its string form. It is used mostly for debugging purposes. The converted string form of the hash table instance is returned as a lengthy String object. This method overrides the toString method in the Object class. The string contains a list of <key, object> pair represented as strings.

**Imports**

*import java.utils.Hashtable;*

**Returns**

The string form of the Hashtable object is returned. Return type is String.

**Example**

Refer to the table() method in the inventory class in Listing 12-1.

## Properties

**Purpose**

Class containing the persistent properties of the associated object.

**Syntax**

public class Properties extends Hashtable

**Description**

Properties subclasses the class Hashtable. This contains the persistent properties of the associated object, whose properties are stored in the hash table. The contents can be saved or loaded from a stream. This class contains a protected member variable named defaults. If a property is not found in the object, it is searched for in the default properties list. This class allows arbitrary nesting of properties. This class is most widely used with the System class to obtain system properties like user name, environment, and so on. Figure 12-8 illustrates the inheritance relationship of class Properties.

**Figure 12-8** Class diagram of Properties class

**PackageName**

*java.util*

**Imports**

*import java.util.Properties;*

**Constructors**

public Properties()

public Properties(Properties *defaults*)

**Parameters**

***defaults***

Initial specified default properties.

**Example**

The propDemo class in Listing 12-2 demonstrates the methods in the Properties class. The Properties object member in the System class in the *java.lang* package is used for demonstration.

**Listing 12-2** propDemo.java: Demonstrating the usage of methods in the Properties class

```
import java.util.*;
import java.io.*;
import java.lang.*;

class propDemo {
  public propDemo() {
  }

  public void printProps() {
       System.out.println(" System key/properties are:\n");
       for (Enumeration e=
  System.getProperties().propertyNames();
  e.hasMoreElements();) {
```

```
                String elem = (String)e.nextElement();
                System.out.println(" " + elem +" " +
                System.getProperty(elem));
        }
  }

  private void getProp(String prop){
        System.out.println("Value of property " +
                    prop + " is " + System.getProperty(prop));

  }

  private void changeProperties(String file_name){

        File file = new File(file_name);
        try {
                System.getProperties().load(new
FileInputStream(file));
        } catch(IOException ioe){
                System.out.println(" Exception reading file");
        }

  }

  private void saveProperties(String file_name){

        File file = new File(file_name);
        String header = new String(" These are changed properties");
        try {

                System.getProperties().save(new FileOutputStream(file),
        header);
        } catch(IOException ioe){
  System.out.println(" Exception writing to file");
        }
  }

  public final static void main(String args[]) {

        propDemo p = new propDemo();
        // list the properties one by one
        p.printProps();

        // Another way to list the properties is thru list() method
        System.out.println("---------------------------------------");
        System.getProperties().list(new PrintStream(System.out));

        System.out.println("---------------------------------------");
        p.getProp("user.dir");
        p.changeProperties("myJava.prop");

        System.out.println("------------ After loading new properties
        -----");
        System.getProperties().list(new PrintStream(System.out));

        p.saveProperties("Changed.prop");
  }
```

```
}
```

## getProperty(String), getProperty(String, String)

**ClassName**

Properties

**Purpose**

Obtain the property with the specified key.

**Syntax**

public String getProperty(String key)
public String getProperty(String key, String defaultValue)

**Parameters**

***key***

The specified key to be searched in the property list.

***defaultValue***

The string to be returned, if the specified key is not found in the list.

**Description**

This method searches for the specified key in the property list. If it is not found in the list, the defaults in the Properties object are searched. If a property is found, it is returned. If you specify the defaultValue during the method invocation, the defaultValue is returned if the property is not found in the list or the defaults. If defaultValue is not specified and if the property is not found either in the property list or the defaults, then the method returns a null.

**Imports**

*import java.utils.Properties;*

**Returns**

The string form of the property searched for. Return type is String.

**Example**

Refer to the getProp method in Listing 12-2. The value of the specified property key is retrieved and printed to the screen using this method.

## list(PrintStream)

**ClassName**

Properties

**Purpose**

The properties are listed for debugging into the specified PrintStream.

**Syntax**

public void list(PrintStream out)

**Parameters**

***out***

The print stream, into which the properties are to be listed.

**Description**

This method lists the properties into the specified print stream. This is used for debugging.

**Imports**

*java.utils.Properties;*

**Returns**

None.

**See Also**

The save method in the Properties class

**Example**

Refer to the main method in Listing 12-2. The list method is used to list the properties of the System object to the screen.

## load(InputStream)

**ClassName**

Properties

**Purpose**

Loads the properties from the specified InputStream.

**Syntax**

public void load(InputStream in)

**Parameters**

***in***

The input stream that the properties are loaded from.

**Description**

This method loads the properties from the specified input stream. If there is an error in reading from InputStream, an IOException is thrown.

***Imports***

*import java.utils.Properties;*

**Returns**

None.

**See Also**

The save method in the Properties class

**Example**

Refer to the changeProperties method in Listing 12-2. The properties defined in file myJava.prop are loaded as the new properties for System object.

## propertyNames()

**ClassName**

Properties

**Purpose**

Enumerates the property keys.

**Syntax**

public Enumeration propertyNames()

**Parameters**

None.

**Description**

All the keys used for properties in the Properties hash table are enumerated. The enumeration that is formed is returned by this method. You can use the enumeration to access the elements sequentially.

**Imports**

*import java.utils.Properties;*

**Returns**

The Enumeration of the keys is returned. Return type is Enumeration.

**See Also**

Enumeration interface

**Example**

Refer to the listAll method in Listing 21-2. All the elements are printed to the screen using the method propertyNames to get the enumeration of the elements.

## save(OutputStream, String)

**ClassName**

Properties

**Purpose**

Saves the properties to the specified OutputStream.

**Syntax**

public void save(OutputStream out, String header)

**Parameters**

***out***

The output stream that the properties are saved to.

***header***

Comment to be used at the top of the output stream object.

**Description**

This method saves the properties to the specified output stream. The specified header is included as a comment at the top of the output file if the output stream is a file stream, or the header is printed at the top as a comment.

**Imports**

*import java.utils.Properties;*

**Returns**

None.

**See Also**

The load method in the Properties class

**Example**

Refer to the saveProperties method in Listing 12-2. The properties of System object are saved to the specified file.

## Vector

**Purpose**

Class representing an extensible array designed for space optimization.

**Syntax**

public class Vector extends Object implements Cloneable

**Description**

Vector is an extensible array designed for storage optimization. Each vector maintains a capacity and capacityIncrement to optimize storage management. The capacity of the Vector is always at least the size of the Vector. Because a vector increases by chunks of memory of size capacityIncrement, the capacity of the vector is usually larger than the vector size. If you specify a greater capacity value during the construction of Vector, then the amount of incremental reallocation will be reduced. This results in better storage management, but if the specified capacity is too large, then storage is wasted. You should decide the size with this in mind. Even if you ignore the capacity while constructing the Vector, it will still work fine. It will simply be less efficient in terms of storage management. This class implements the Cloneable interface. So the clone()method is defined in it. The Vector class has three variables: *elementData*, *elementCount*, and *capacityIncrement*. The *elementData* variable is the buffer where elements are stored. The elementCount variable contains the number of elements in the buffer. The capacityIncrement variable contains the size of the increment. Figure 12-9 illustrates the inheritance relationship of the Vector class.

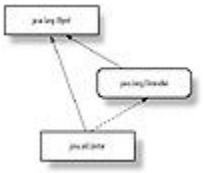

**Figure 12-9** Class diagram of Vector class

**PackageName**

*java.util*

**Imports**

*import java.util.Vector;*

**Constructors**

public Vector()
public Vector(int *init_capacity*)
public Vector(int *init_capacity*, int *capacityIncrement*)

**Parameters**

***init_capacity***

Initial specified capacity of the Vector.

***capacityIncrement***

The size of the increment. If it is 0 then the capacity is doubled every time the vector needs to grow.

**Variables**

protected Object *elementData[]*
protected int *elementCount*
protected int *capacityIncrement*

**Example**

The guestBook class, in Listing 12-3, is used to illustrate the usage of the Vector class to create an object whose size is not known in advance and whose size will vary with time. The member variable *guests* is an object of type Vector.

**Listing 12-3** guestBook.java: A guest book maintained using the Vector class to keep log of guests

```
import java.util.*;

class guestBook {

 Vector guests;

 public guestBook() {

        guests = new Vector(5, 4);
 }

 private void initialGuests() {
        guests.addElement("Anne");
        guests.addElement("Bob");
        guests.addElement("Chuck");
        guests.addElement("David");
        guests.addElement("Emy");
        guests.addElement("Fred");
 }

 private int bookCapacity() {
        return guests.capacity();
 }

 private boolean visited(Object guest){
        return guests.contains(guest);
 }

 private Object guestNo(int number){
        return guests.elementAt(number);
 }

 protected void listAll() {
         System.out.println(" Guests visited are: ");
         for (Enumeration e = guests.elements(); e.hasMoreElements();){
                 System.out.println(" " + e.nextElement());
         }
        /*      String str[] = new String[3];
        guests.copyInto(str); */
 }

 protected void minSize(int min) {
         System.out.println("Capacity before setting min " +
         guests.capacity());
         guests.ensureCapacity(min);
         System.out.println("Capacity after setting min is "
         +  guests.capacity());

 }
 private void analyze() {
        System.out.println(" First guest is " +
```

```
        guests.firstElement());
        System.out.println(" David came after guest #" +
        guests.indexOf("David"));
        int no = guests.lastIndexOf("Emy");
        System.out.println(" Emy came after " +
        guests.elementAt(no-1));
        System.out.println(" Last guest at this time is
        "+guests.lastElement());

}
private void replace(Object g1, Object g2) {
        int index = guests.indexOf(g1);
        guests.removeElementAt(index);
        if (!guests.removeElement(g1))
                System.out.println(g1 + "is no more in the guest log");

        guests.insertElementAt(g2, index);
}
private void change(Object g1, Object g2){
        int index = guests.indexOf(g1);
        guests.setElementAt(g2, index);
}

Vector backup;

protected void makeBackup() {
        backup = (Vector)guests.clone();
        for (Enumeration e = guests.elements(); e.hasMoreElements();){
                backup.addElement(e.nextElement());
       }
}
private void newcapacity(int cap){
        guests.setSize(4);
        System.out.println(" new capacity/size is
        "+guests.capacity()+"/"+guests.size());
        guests.trimToSize();
        System.out.println(" new capacity/size is
        "+guests.capacity()+"/"+guests.size());
        }
        public final static void main(String args[]) {
        guestBook gb = new guestBook();
        // gb.analyze();
        gb.initialGuests();
        System.out.print("after adding 6 guests, the book
        capacity is ");
        System.out.println(gb.bookCapacity());
        System.out.print(" Bob is ");
        if (!gb.visited("Bob"))
            System.out.print("not ");
        System.out.println("one of the guests");

        System.out.println(" The third guest is " +
        gb.guestNo(2));

        gb.listAll(); /* list all the guests visited till now */
        gb.minSize(10);
```

```
        gb.analyze();
        gb.replace("Chuck", "Charlie");
        gb.listAll();
        gb.change("Emy", "Ellis");
        gb.listAll();

        gb.newcapacity(4);

 }
}
```

## addElement(Object)

**ClassName**

Vector

**Purpose**

Adds the specified element to the Vector.

**Syntax**

public final synchronized void addElement(Object element)

**Parameters**

***element***

The object to be added to the vector.

**Description**

This method adds the specified element to the end of the vector. The added element becomes the last element in the vector. If the size of the vector is equal to the capacity of the vector before adding the element, then the size of the vector is increased by the size of capacityIncrement.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**See Also**

The insertElementAt method in the Vector class

**Example**

Refer to the initialGuests method in Listing 12-3. Guests are added to the guests Vector using this addElement method.

## capacity()

**ClassName**

Vector

**Purpose**

Obtains the current capacity of the vector.

**Syntax**

public final int capacity()

**Parameters**

None.

**Description**

This method gets the current capacity of the vector.

**Imports**

*import java.utils.Vector;*

**Returns**

The current capacity of the vector; the return type is int.

**See Also**

The ensureCapacity method in the Vector class

**Example**

Refer to the bookCapacity method in Listing 12-3. The capacity method is used to return the capacity of the guests vector.

## clone()

**ClassName**

Vector

**Purpose**

Clones the target Vector object.

**Syntax**

public synchronized Object clone()

**Parameters**

None.

**Description**

This method creates a clone of the vector and returns a handle to it. A shallow copy of the vector is made with its characteristics. None of the elements of the vector are cloned in this operation. This cloning operation is a relatively expensive operation. This method overrides the clone method of the Object class. For further operations, typecast the Object returned to Vector type. This method is defined as the Vector class implements the Cloneable interface

**Imports**

*import java.utils.Vector;*

**Returns**

A handle to the cloned instance of the Vector. Return type is Object, which you can typecast to Vector.

**See Also**

The insertElementAt method in the Vector class.

**Example**

Refer to the makeBackup method in Listing 12-3. A new copy of the member guests is made by using the clone method on the guests object. This forms the backup member.

## contains(Object)

**ClassName**

Vector

**Purpose**

Checks if the specified object is an element in the Vector.

**Syntax**

public final boolean contains(Object elem)

**Parameters**

***elem***

The element that we are searching for.

**Description**

This method checks if the specified element is in the Vector. It returns true if the object is in the vector, false if it is not. If the specified value is null, then this method throws a NullPointerException.

**Imports**

*import java.utils.Vector*;

**Returns**

If the object is in the vector, the method returns true. If the object is not in the vector, the method returns false. Return type is boolean.

**Example**

Refer to the visited method in Listing 12-3. It verifies if the specified guest has visited, by checking if the name is in the guests vector. This is implemented using the contains method on the vector guests.

## copyInto(Object[])

**ClassName**

Vector

**Purpose**

Copies the elements of the vector into the specified array.

**Syntax**

public final synchronized void copyInto(Object newArray[])

**Parameters**

***newArray***

The array that the elements of the vector are copied into.

**Description**

This method copies the elements in the vector into the specified array. Note that the clone() method creates a vector and does not copy the elements. This method copies the elements into the array. If the array size is less than the size of the vector, an ArrayIndexOutOfBoundsException is thrown.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**Example**

Refer to the listAll method in Listing 12-3. If you uncomment the lines where the elements are copied into a three element array, an exception will be thrown.

## elementAt(int)

**ClassName**

Vector

**Purpose**

Obtain the element in the vector at the specified index.

**Syntax**

public final synchronized Object elementAt(int index)

**Parameters**

***index***

The index position of the element that this method retrieves.

**Description**

This method gets the element at the specified index in the vector and returns the element. If the specified index is more than the size of the vector, then it throws an ArrayIndexOutOfBoundsException.

**Imports**

*java.utils.Vector;*

**Returns**

The element at the specified index in the target vector object. Return type is Object.

**See Also**

The setElementAt method in the Vector class

**Example**

Refer to the guestNo method in Listing 12-3. It returns the guest name which is the specified numbered person to visit. It uses the elementAt method of the Vector class to get the required name.

## elements()

**ClassName**

Vector

**Purpose**

Obtains an enumeration of the elements.

**Syntax**

public final synchronized Enumeration elements()

**Parameters**

None.

**Description**

This method returns an enumeration of the elements in a vector. The enumeration that is returned can be used to fetch the elements sequentially from the vector.

**Imports**

*import java.utils.Vector;*

**Returns**

An Enumeration of elements in the vector. Return type is Enumeration.

**See Also**

The Enumeration interface

**Example**

Refer to the listAll method in Listing 12-3. The enumeration of the guests is obtained by using this elements() method.

## ensureCapacity(int)

**ClassName**

Vector

**Purpose**

Verifies if the Vector has at least the specified capacity.

**Syntax**

public final synchronized void ensureCapacity(int minCapacity)

**Parameters**

***minCapacity***

The desired minimum capacity of the vector.

**Description**

The method makes sure that the target Vector object has at least the specified minimum capacity. If so, it does nothing. If the vector capacity is less than the minimum capacity, then the capacity of the vector is increased to the desired minimum.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**See Also**

The capacity method in the Vector class

**Example**

Refer to the minSize method in Listing 12-3, which ensures that the guest book capacity is at least the specified minimum.

## firstElement()

**ClassName**

Vector

**Purpose**

Obtains the first element in the vector.

**Syntax**

public final synchronized Object firstElement()

**Parameters**

None.

**Description**

This method returns the first element in the sequence of the target Vector object. If the sequence is empty, this method throws a NoSuchElementException.

**Imports**

import java.utils.Vector;

**Returns**

The first element in the vector. Return type is Object.

**See Also**

The lastElement method in the Vector class.

**Example**

Refer to the analyze method in Listing 12-3. The first visitor is found by using the firstElement method on the guests vector.

## indexOf(Object), indexOf(Object, int)

**ClassName**

Vector

**Purpose**

Obtains the index of the specified object in the vector.

**Syntax**

public final int indexOf(Object element)
public final int indexOf(Object element, int startFrom)

**Parameters**

***element***

The object to be searched for in the vector.

***startFrom***

The index location from which to start searching for the element in the vector.

**Description**

This method searches for the specified element and returns the index position of the element in the vector, if the element is in the vector. If the element is not in the vector, an ArrayIndexOutOfBoundsException is thrown. If you specify the index to start the search from, as a part of the method call, then the search starts from the specified index in the vector.

**Imports**

*import java.utils.Vector;*

**Returns**

The index position of the searched element in the vector.

**See Also**

The lastIndexOf(Object) and lastIndexOf(Object, int) methods in the class Vector

**Example**

Refer to the analyze method in Listing 12-3. The number of guests who arrived after David is ascertained by using the index of David's entry in the guests vector. The indexOf method is used to get the information.

## insertElementAt(Object, int)

**ClassName**

Vector

**Purpose**

Inserts the specified element at the specified index.

**Syntax**

public final synchronized void insertElementAt(Object element, int index)

**Parameters**

***element***

The element to be inserted into the vector.

***index***

The index location where the specified element is to be inserted.

**Description**

This method inserts the specified element at the specified index in the vector. The indices of the elements existing at and after the index, before this insertion, are shifted accordingly and their indices are incremented by one. If the specified index is invalid, then this method throws an ArrayIndexOutOfBoundsException.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**See Also**

The removeElementAt method in the Vector class

**Example**

Refer to the replace method in Listing 12-3. After removing the specified object, the new object is placed at its position using the insertElementAt method of the Vector class.

## isEmpty()

**ClassName**

Vector

**Purpose**

Checks to see if the Vector contains any elements.

**Syntax**

public final boolean isEmpty()

**Parameters**

None.

**Description**

This method returns true if the Vector contains no elements. If the vector contains even one element, this method returns false.

**Imports**

*import java.utils.Vector;*

**Returns**

True if empty and false if the Vector contains one or more elements; return type is boolean.

**See Also**

The elements method in the Vector class

**Example**

Refer to Listing 12-3. You can check if anyone visited by using this isEmpty method on the guests vector member. It will return true if no guests visited and false if at least one guest is in the guests vector.

## lastElement()

**ClassName**

Vector

**Purpose**

Returns the last element in the vector.

**Syntax**

public final synchronized Object lastElement()

**Parameters**

None.

**Description**

This method returns the last element in the sequence of the target Vector object. If the sequence is empty, this method throws a NoSuchElementException.

**Imports**

*import java.utils.Vector;*

**Returns**

The last element in the vector. Return type is Object.

**See Also**

The firstElement method in the Vector class

**Example**

Refer to the analyze method in Listing 12-3. The last guest to visit is found by using the lastElement method.

## lastIndexOf(Object), lastIndexOf(Object, int)

**ClassName**

Vector

**Purpose**

Obtains the index of the specified object in the vector by searching the object backwards.

**Syntax**

public final int lastIndexOf(Object element)
public final int lastIndexOf(Object element, int startFrom)

**Parameters**

***element***

The object that the vector is to be searched for.

***startFrom***

The index location from which to start searching for the element in the vector.

**Description**

This method searches backward from the end of the vector for the specified element and returns the index position of the element in the vector, if the element is in the vector. If the element is not in the vector, it returns -1. If you specify the index to start the search from as part of the method call, then the search starts from the specified index and moves backward in the vector.

**Imports**

*import java.utils.Vector;*

**Returns**

The index position of the searched element in the vector if the element is found; if the element is not found, it returns -1.

**See Also**

The firstIndexOf(Object) and firstIndexOf(Object, int) methods in the Vector class

**Example**

Refer to the analyze method in the guestBook class in Listing 12-3. The visitor number of Emy is found by using this method to scan the vector from the rear.

## removeAllElements()

**ClassName**

Vector

**Purpose**

Removes all the elements in the vector.

**Syntax**

public final synchronized void removeAllElements()

**Parameters**

None.

**Description**

The method removes all the elements in the vector, leaving the vector empty.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**See Also**

The removeElement and removeElementAt methods in the Vector class

**Example**

Refer to Listing 12-3. You can clear all the names of the guests by invoking the removeAllElements method on the vector member, guests.

## removeElement(Object)

**ClassName**

Vector

**Purpose**

Removes the specified element from the vector.

**Syntax**

public final synchronized boolean removeElement(Object element)

**Parameters**

***element***

The element to be removed from the vector.

**Description**

This method removes the specified element in the vector. If the element is not found in the vector, this method returns false. If the vector contains two or more

occurrences of the element, only the first occurrence is removed. If the element is found in the vector, the method returns true after removing the element.

**Imports**

*import java.utils.Vector;*

**Returns**

If the element is found, this method returns true after removing the element. If the element is not found in the vector, this method returns false.

**See Also**

The removeAllElements and removeElementAt methods in the Vector class

**Example**

Refer to the replace method in Listing 12-3. After removing the guest g1 from its index position, any other instance of a guest with that name is removed or the method reports that no guest with the given name is present.

## removeElementAt(int)

**ClassName**

Vector

**Purpose**

Removes the element at the specified index from the vector.

**Syntax**

public final synchronized void removeElementAt(int index)

**Parameters**

***index***

The index of the element to be removed from the vector.

**Description**

This method removes the element at the specified index in the vector. If the specified index is more than the capacity of the vector, the method throws ArrayIndexOutOfBoundsException. After deleting the element at the specified index, elements with an index greater than the specified index are decremented by one. This effectively moves the set of elements down the vector by one index position.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**See Also**

The removeAllElements and removeElement methods in the Vector class

**Example**

Refer to the replace method in Listing 12-3. The index of guest g1 is first found and then the method removeElementAt is used to remove the guest name from the determined index position.

## setElementAt(Object, int)

**ClassName**

Vector

**Purpose**

Sets the element at the specified index in the vector to the specified object.

**Syntax**

public final synchronized void setElementAt(Object elem, int index)

**Parameters**

***elem***

The specified object that is placed at the specified index in the vector.

***index***

The index in the vector where the new object is to be placed.

**Description**

This method sets the element at the specified index to the specified object. The element previously at that index is replaced by the new specified object. If the specified index is invalid, then ArrayIndexOutOfBoundsException is thrown at runtime.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**Example**

Refer to the change method in Listing 12-3. It replaces guest g1 with guest g2. Though the functionality is the same as that of the replace method, the change method implements it in a different way by setting the element at the determined index location using the setElementAt method.

## setSize(int)

**ClassName**

Vector

**Purpose**

Sets the size of the vector to the specified size.

**Syntax**

public final synchronized void setSize(int newSize)

**Parameters**

***newSize***

New size that the vector is set to.

**Description**

This method sets the size of the vector to the specified newSize. If the specified newSize is more than the size of the vector before the method is invoked, then the extra elements in the vector are set to null. Whereas, if the specified newSize is less than the size of the method before the method is invoked, then the vector is shrunk to the specified newSize and the excess elements are discarded.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**See Also**

The size method in the Vector class

**Example**

Refer to the newcapacity method in Listing 12-3. The specified capacity is set to be the new capacity by using the setSize method on the guests vector object.

## size()

**ClassName**

Vector

**Purpose**

Obtains the size of the vector.

**Syntax**

public final int size()

**Parameters**

None.

**Description**

This method returns the size of the vector; that is, the number of elements in the vector and not the capacity of the vector. The capacity of the vector can be greater than the size of the vector. Whenever the size is about to exceed the capacity of the vector, the capacity is incremented by an amount contained in capacityIncrement.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**See Also**

The capacity and setSize methods in the Vector class

**Example**

Refer to the newcapacity method in Listing 12-3. The size before and after trimming the capacity to specified size is obtained using the size method.

## toString()

**ClassName**

Vector

**Purpose**

Converts the instance of Vector to its string form.

**Syntax**

public final synchronized String toString()

**Parameters**

None.

**Description**

The method converts the target Vector object to its string form, used mostly for debugging purposes. The converted string form is returned as a lengthy String object. This method overrides the toString method in class Object.

**Imports**

*import java.utils.Vector;*

**Returns**

The string form of the Vector object is returned. Return type is String.

**Example**

Refer to Listing 12-3. You can invoke guests.toString() in any of the member methods in the guestBook class. It will convert the guests vector to string form containing the key-element pair of the vector.

## trimToSize()

**ClassName**

Vector

**Purpose**

Reduces the capacity of the vector to the size of the vector.

**Syntax**

Public final synchronized void trimToSize()

**Parameters**

None.

**Description**

This method reduces the capacity of the vector to the size of the vector. At any point, there can be excess allocated space in the vector depending on the capacityIncrement value and size of the vector. Using this method, the excess allocated space can be reclaimed for storage management. After using this method, space is reallocated for any new insertions into the vector.

**Imports**

*import java.utils.Vector;*

**Returns**

None.

**See Also**

The capacity, size, and setSize methods in the Vector class.

**Example**

Refer to the newcapacity method in Listing 12-3. The size of the guests vector is trimmed to the capacity by using this trimToSize method. The changes resulting from the use of this method are observed from the information printed before and after the change.

## Stack

**Purpose**

Class encapsulating the Last-In-First-Out(LIFO) stack of objects.

**Syntax**

public class Stack extends Vector

**Description**

Stack is the encapsulation of a Last-In-First-Out stack of objects. The object inserted last is at the top of the stack. It can be retrieved using the pop method. An object is put on the stack using the push method. Elements not at the top of the stack cannot be accessed until all the objects above it in the stack are popped out. Figure 12-10 illustrates the inheritance relationship of class Stack.

**Figure 12-10** Class diagram of Stack class

**PackageName**

*java.util*

**Imports**

*import java.util.Stack;*

**Constructors**

public Stack()

**Parameters**

None.

**Example**

The class stackDemo in Listing 12-4 has a member *st*, of type Stack.

**Listing 12-4** stackDemo.java: Demonstrating the usage of the Stack class and its methods

```
import java.util.*;

class stackDemo {

 Stack st;

 public stackDemo() {
       st = new Stack();
 }

 public void populateStack(){
       st.push("one");
       st.push("two");
       st.push("three");
       st.push("four");
       st.push("five");
 }

 public void check(Object obj){
       System.out.print(" The object " + obj + " is ");
       if (st.search(obj) == -1)
               System.out.print(" not ");
       System.out.println(" in the stack ");
```

```
 }

 private void replaceTop(Object obj){
        System.out.println(" Replacing the object " + st.peek() +
        " with "+obj);
        st.pop();
        st.push(obj);
 }

 private void clearAll() {
        int size = st.size();
        for (int i =0; i<size; i++)
                st.pop();
        if (st.empty())
                System.out.println(" After popping all elements
        off stack
        the stack is empty");
 }

 public final static void main(String args[]){
       stackDemo sd = new stackDemo();
       // sd.replaceTop("try");
       sd.populateStack();
       sd.check("two");
       sd.check("seven");
       sd.replaceTop("six");
       sd.clearAll();
 }
}
```

## empty()

**ClassName**

Stack

**Purpose**

Boolean value indicating if there is any object on the stack.

**Syntax**

public boolean empty()

**Parameters**

None.

**Description**

This method returns true if the Stack contains no objects. If the stack contains even one object, this method returns false.

**Imports**

*import java.utils.Stack;*

**Returns**

True if the Stack is empty and false if the Stack contains one or more objects.

**Example**

Refer to the clearAll method in Listing 12-4. After clearing all the elements from the stack, this method verifies that the stack is empty.

## peek()

**ClassName**
Stack
**Purpose**
Peeks at the element at the top of the stack.
**Syntax**
public Object peek()
**Parameters**
None.
**Description**
This method obtains the object at the top of the stack. This method does not affect the elements on the stack. The top object on the stack remains on the stack, but only a copy of it is returned. If the stack contains no element, this method throws EmptyStackException at runtime.
**Imports**
*import java.utils.Stack;*
**Returns**
A copy of the object at the top of the stack, if the stack is not empty.
**See Also**
The pop method in the Stack class
**Example**
Refer to the replaceTop method in Listing 12-4. In this method, the top element is replaced by a specified element. Before popping out the top element, the peek method is used to find the top element.

## pop()

**ClassName**
Stack
**Purpose**
Obtains the element at the top of the stack.
**Syntax**
public Object pop()
**Parameters**
None.
**Description**
This method obtains the object at the top of the stack. This method removes the object from the top of the stack and returns it, leaving the stack with one less element. If the stack is empty, this method throws an EmptyStackException during runtime.
**Imports**
*import java.utils.Stack;*
**Returns**
The object at the top of the stack, if the stack is not empty.
**See Also**

The peek and push methods in the Stack class

**Example**

Refer to the replaceTop method in the stackDemo class in Listing 12-4. The top element in the stack is popped off using this method before the specified element is pushed onto the stack.

## push(Object)

**ClassName**

Stack

**Purpose**

Pushes the specified object onto the stack.

**Syntax**

public Object push(Object obj)

**Parameters**

***obj***

The object to be pushed onto the stack.

**Description**

This method pushes the specified object onto the stack. The object is placed on top of the stack. If the pop method is invoked, the last inserted object will be popped off the stack. This method places the object onto the stack and returns a handle to the object

**Imports**

*import java.utils.Stack;*

**Returns**

The object at the top of the stack, after placing it at the top.

**See Also**

The pop method in the Stack class

**Example**

Refer to the populateStack method in Listing 12-4. Elements are added to the stack member by using the push method of class Stack.

## search(Object)

**ClassName**

Stack

**Purpose**

Searches for the specified object in the stack.

**Syntax**

public int search(Object obj)

**Parameters**

***obj***

The object to be searched in the stack.

**Description**

This method searches for the specified object in the stack. If the object is found, its index position from the top of the stack is returned. If the object is not found,

this method returns -1. This method is useful to search for items within the stack without affecting the stack structure.

**Imports**

*import java.utils.Stack;*

**Returns**

The index location of the object searched for in the stack. If it is not found, a value of -1 is returned.

**Example**

Refer to the check method in Listing 12-4. The presence, in the stack, of the specified object is verified by using the search method in class stack.

## Enumeration

**Purpose**

Interface specifying methods to enumerate a set of objects.

**Syntax**

public interface Enumeration extends Object

**Description**

The Enumeration interface specifies a set of methods: hasMoreElements() and nextElement(). They are used to enumerate or count through a set of values. The enumeration is consumed as you use it to access the elements and you cannot get back the used enumeration. So the elements can be counted only once. You have to obtain the enumeration again if you want to access an enumeration earlier accessed. Figure 12-11 illustrates the inheritance relationship of interface Enumeration.

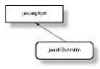

**Figure 12-11** Interface diagram of Enumeration

**PackageName**

*java.util*

**Imports**

*import java.util.Enumeration;*

**Example**

Refer to Listing 12-4. All the guests are listed in the listAll method. This is done by first getting the enumeration of guests by using the elements() method of class Vector.

## hasMoreElements()

**Interface**

Enumeration

**Purpose**

Determines if the enumeration contains more elements.

**Syntax**

public abstract boolean hasMoreElements()

**Parameters**

None.

**Description**

This method checks if the enumeration has more elements. It returns true if the enumeration has more elements, and returns false if the enumeration does not have more elements. This is an abstract method, as it is in an interface and it should be defined in the class implementing the interface Enumeration.

**Imports**

*import java.utils.Enumeration;*

**Returns**

The boolean value true is returned if the enumeration has more elements. If the enumeration is empty, the method returns false. Return type is boolean.

**Example**

Refer to the listAll method in Listing 12-4. All the guests are listed by getting the enumeration of elements in vector guests. This method, hasMoreElements, is used on the enumeration in the for loop construct while printing all the guest names.

## nextElement()

**Interface**

Enumeration

**Purpose**

Returns the next element in the enumeration.

**Syntax**

public abstract Object nextElement()

**Parameters**

None.

**Description**

The method returns the next element in the enumeration. This method enumerates successive elements. If no more elements exist in the enumeration and this method is invoked, then a NoSuchElementException is thrown. This is an abstract method as it is a method in an interface and it should be defined in the class implementing the interface Enumeration.

**Imports**

*import java.utils.Enumeration;*

**Returns**

The successive object in the enumeration is returned. Return type is Object.

**Example**

Refer to the method listAll in Listing 12-4. All the guests are listed by getting the enumeration of elements in vector guests. This method, nextElement, is used on the enumeration in the for loop construct while printing all the guest names.

## EmptyStackException

**Purpose**

A Runtime exception signaling an empty stack.

**Syntax**

public class EmptyStackException extends RuntimeException

**Description**

If a method tries to access an element in an empty stack, EmptyStackException is thrown. For example, it is thrown when you try to use pop() method on an empty stack. It is a runtime exception and hence, need not be declared to be thrown. The constructor of this exception does not have any detailed message associated with it that describes the exception. Figure 12-12 illustrates the inheritance relationship of class EmptyStackException.

**Figure 12-12** Class diagram of EmptyStackException class

**PackageName**

*java.util*

**Imports**

*import java.util.EmptyStackException;*

**Constructors**

public EmptyStackException()

**Parameters**

None.

**Example**

If you try to replace the top of the stack using the replaceTop method before populating the stack, it will throw an EmptyStackException. Try uncommenting the code before populate method in the main method in Listing 12-4.

## NoSuchElementException

**Purpose**

A Runtime exception signaling an empty enumeration.

**Syntax**

public class NoSuchElementException extends RuntimeException

**Description**

If a method tries to access an element in an empty enumeration, NoSuchElementException is thrown. For example, it is thrown when you try to use the firstElement() or lastElement() methods on an empty enumeration sequence. It is a runtime exception and hence, need not be declared to be thrown. The constructor of this exception does not have any detailed message associated with it that describes the exception. Figure 12-13 illustrates the inheritance relationship of class NoSuchElementException.

**Figure 12-13** Class diagram of NoSuchElementException class

**PackageName**
*java.util*
**Imports**
*import java.util.NoSuchElementException;*
**Constructors**
public NoSuchElementException()
**Parameters**
None.
**Example**
Refer to Listing 12-3. If you call the analyze method before the vector is populated (when it is empty), the NoSuchElementException results when the firstElement is accessed. If you uncomment the second line in the main method in that listing, this exception is thrown.

## Random

**Purpose**
A pseudo-random number generator class.
**Syntax**
public class Random extends Object
**Description**
The Random class is used to generate a stream of pseudo-random numbers. One of the constructors is used to create a new random number generator. You can specify a seed of type long or a random number generator with no seed. When you specify a seed, the same stream of pseudo-random numbers can be repeated using the same seed to start the generator. If you don't specify a seed, a value based on the current time is used as a seed. If you want to reset the seed to a different value, you can use the setSeed method. To obtain successive pseudo-random numbers, you can use one of the other methods in this class (except the setSeed method). Figure 12-14 illustrates the inheritance relationship of class Random.

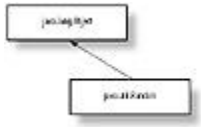

**Figure 12-14** Class diagram of Random class

**PackageName**

*java.util*

**Imports**

*import java.util.Random;*

**Constructors**

public Random()

public Random(long *seed*)

**Parameters**

***seed***

The single long seed used to create a random number generator.

**Example**

The randomDemo class in Listing 12-5 illustrates the generation of random numbers.

**Listing 12-5** randomDemo.java: Usage of random numbers

```
import java.util.Random;

class randomDemo {

  public randomDemo() {
 }

 private void doubleGen() {
       Random r = new Random(10);
       for (int i=0;i<5; i++)
               System.out.print(r.nextDouble() + " ");
       System.out.println("\n\n");
 }
 private void floatGen() {
       Random r = new Random();
       for (int i=0;i<5; i++) {
               float f = r.nextFloat();
               if (f >0.5)
                       System.out.print(f + " ");
       }
       System.out.println("\n\n");
 }
 private void gaussianGen() {
       Random r = new Random(10);
       for (int i=0;i<5; i++)
               System.out.print(r.nextGaussian() + " ");
       System.out.println("\n\n");
 }

 private void intGen() {
       Random r = new Random();
       for (int i=0;i<5; i++) {
               int num = r.nextInt();
               if (num > 0)
                       System.out.print(num + " ");
       }
       System.out.println("\n\n");
```

```
 }
 private void longGen() {
       Random r = new Random();
       r.setSeed(300);
       for (int i=0;i<5; i++) {
               long l = r.nextLong();
               if ( l < 1000000)
                       System.out.print(l + " ");
       }
       System.out.println("\n\n");
 }

 public final static void main(String args[]) {
       randomDemo rd = new randomDemo();

       System.out.println(" Random generator of numbers of type
Double\n");
       rd.doubleGen();

       System.out.println(" Random generator of numbers of type
Float\n");
       rd.floatGen();

       System.out.println(" Random generator of numbers of type
(Gaussian)
       double\n");
       rd.gaussianGen();

       System.out.println(" Random generator of numbers of type
Int\n");
       rd.intGen();

       System.out.println(" Random generator of numbers of type
Long\n");
       rd.longGen();

 }
}
```

## nextDouble()

**ClassName**

Random

**Purpose**

Generates a pseudo-random uniformly distributed double value between 0.0 and 1.0.

**Syntax**

public double nextDouble()

**Parameters**

None.

**Description**

This method generates a pseudo-random number, uniformly distributed double value between 0.0 and 1.0.

**Imports**

*import java.utils.Random;*

**Returns**

A double value between 0.0 and 1.0.

**Example**

Refer to the doubleGen method in Listing 12-5. Random double values are generated with seed 10. For any number of runs of the program, the same sequence of double values is generated.

## nextFloat()

**ClassName**

Random

**Purpose**

Generates a pseudo-random uniformly distributed float value between 0.0 and 1.0.

**Syntax**

public float nextFloat()

**Parameters**

None.

**Description**

This method generates a pseudo-random number, uniformly distributed float value between 0.0 and 1.0.

**Imports**

*import java.utils.Random;*

**Returns**

A float value between 0.0 and 1.0.

**Example**

Refer to the floatGen method in Listing 12-5. The float values above 0.5 are printed to the screen. Each time this program is run, the sequence differs. This is because the Random object is not created with a fixed seed, as in the case of doubleGen.

## nextGaussian()

**ClassName**

Random

**Purpose**

Generates a pseudo-random, Gaussian, distributed double value between 0.0 and 1.0.

**Syntax**

public double nextGaussian()

**Parameters**

None.

**Description**

This method generates a pseudo-random number, Gaussian, distributed double value with mean 0.0 and standard deviation 1.0.

**Imports**

*import java.utils.Random;*

**Returns**

A double value.

**Example**

Refer to the gaussianGen method in Listing 12-5. It generates values of type double. As you will note, it can generate negative values, whereas, the doubleGen method generates only positive double. This illustrates the difference between nextDouble and nextGaussian, which differs not only in distribution but also in possible values generated.

## nextInt()

**ClassName**

Random

**Purpose**

Generates a pseudo-random uniformly distributed int value.

**Syntax**

public int nextInt()

**Parameters**

None.

**Description**

This method generates a pseudo-random number, uniformly distributed int value.

**Imports**

*import java.utils.Random;*

**Returns**

An int value.

**Example**

Refer to the inGen method in Listing 12-5. Positive integer values are printed to the output stream. In this method, the generated sequence of numbers varies for every run of the program.

## nextLong()

**ClassName**

Random

**Purpose**

Generates a pseudo-random uniformly distributed long value.

**Syntax**

public long nextLong()

**Parameters**

None.

**Description**

This method generates a pseudo-random number uniformly distributed long value.

**Imports**

*import java.utils.Random;*

**Returns**

A long value.

**Example**

Refer to the longGen method in Listing 12-5. The sequence of numbers generated remains the same because the seed is set to 300 using the setSeed function. Those numbers generated below 1 million are printed to the output stream.

### setSeed(long)

**ClassName**

Random

**Purpose**

Sets the seed of the random number generator using the specified single long seed.

**Syntax**

public synchronized void setSeed(long seed)

**Parameters**

***seed***

The seed for the pseudo-random number generator.

**Description**

This method sets the seed for the pseudo-random number generator to the specified seed. This method can also be used to reset earlier seed values. A random number generator with a specified seed generates a repeatable stream of pseudo-random numbers.

**Imports**

*import java.utils.Random;*

**Returns**

None.

**Example**

Refer to the longGen method in Listing 12-5. The sequence of numbers generated remains the same bacause the seed is set to 300 using the setSeed function. Generated numbers below 1 million are printed to the output stream.

## Java Appointment Organizer Applet

The applet for this chapter is an appointment organizer. Using Java's AWT components you will provide the user-interface for entering appointment details. The user will enter a date, time, and a text string describing the appointment. Users can save the appointment information and then retrieve it by specifying the data and the time. To develop the user-interface for the organizer, you'll create the following components:

1. A text field for entering the date
2. A text field for entering the hour of appointment

**3.** A text field for entering the minutes past the hour at which the appointment is scheduled
**4.** A choice button to select between setting an appointment and finding the appointment at a given date and time
**5.** A text area to enter appointment details
**6.** A text area to display the details of the requested appointment
**7.** A button to confirm the user's selection of setting or finding an appointment. Appropriate action will be taken after this confirmation button is clicked.

The Java Appointment Organizer is constructed using the classes covered in this chapter. You will create a javaOrganizer class with member dates. The javaOrganizer class will subclass the Applet class, because we are building an applet. The member dates will be a dateVector object with a capacity of 31 elements. The class dateVector is a subclass of Vector. Date will be used as an index into the dateVector to map to an appointment on a given date. In this implementation, we'll consider only the present month. You can build on this design to implement for 12 months a year and for any specified year. Each object in the vector, named dates, is a hash table because the time and number of appointments will vary from day to day. On some days a person might have many appointments and on others very few. Appointments are arranged with time as the key element and description as the object. For a given index in the dateVector, an appointmentTable object is stored. The appointmentTable class is a hash table class.

The user can trigger two actions: setting up a new appointment and retrieving an appointment at a given date and time. The javaOrganizer includes methods to accomplish these two actions. Methods in the Vector, HashTable and Date classes provide the necessary functionality expected from the javaOrganizer.

## Building the Project

**1.** Using the bottom-up approach, we shall first design the appointmentTable class. It is a subclass of the Hashtable class. We will specify the capacity of the hash table to be 10. In case you want to include more time, this capacity is expandable. Enter the following code to define the appointmentTable class.

```
class appointmentTable extends Hashtable {

        public appointmentTable() {
                super(10);
                        // 10 hours a day to start with
        }
}
```

**2.** The appointmentTable object is a part of the dates vector. We need to add functionality for this class to add appointment details for a given time. Date is determined by the index occupied by this appointmentTable object in the dates vector. We will add a method addOneAppt. This method will take two parameters: time and details. The time is the key for this table. This method will add the given appointment details with the specified key to the table. If there is already another appointment at the specified time, this method will notify the user and return without affecting the earlier appointment. Another needed functionality

is the ability to retrieve an appointment detail text, given the time. The apptAt method will return the String containing the details when the time is passed as a parameter. Add the following code to the appointTable class, defined in Step 1.

```
public boolean addOneAppt(String time, String what)
{
    if (this.containsKey(time)) {
        System.out.println("An appt at " + time
                + " already exists!");
        return false;
     }

     put(time, what);
     return true;
}

public String apptAt(String time) {

   if (!this.containsKey(time)) {
       System.out.println("You have no appt at" + time);
       return null;
   }
   return (String)this.get(time);
}
```

**3.** Having defined the appointmentTable class, we will now define the dateVector class, which will subclass the Vector class. This class should have the functionality to add and retrieve appointment details. It is a Vector object with a size of 31, so we can index into 31 locations into the vector. Without much optimization, the following code defines the class and its constructor. We populate all its indices with default appointmentTable objects, so that there is a table for each day of the month.

```
class dateVector extends Vector {

   public dateVector() {
           super(31); // vector for 31 days
           for (int i =0; i <31; i ++)
                 this.addElement(new appointmentTable());
       }
}
```

**4.** Now we'll add methods to the dateVector class to add and retrieve appointment tables. Each appointmentTable object is indexed by date. To add a specified appointmentTable on a particular day, we will define a fixAppt method. It will take a date as the index into the vector and the appointmentTable as the element. The knowAppt method will find the appointment table for the specified date. It returns an object of type appointmentTable. These methods are defined here. Enter the following code into the dateVector class defined in the previous step.

```
public void fixAppt(appointmentTable appt, int date) {
        if (date > 31 || date <1) {
            System.out.print(" Error in adding Appt -");
            System.out.println(" Invalid date " + date );
            return;
        }
        this.setElementAt(appt, date-1);
```

```
}

public appointmentTable knowAppt(int date){
        if (date > 31 || date <1) {
            System.out.println(" Error in getting Appt - ");
            System.out.println(" Invalid date " + date );
            return null;
        }
        return (appointmentTable)elementAt(date-1);
}
```

**5.** Now that you have defined two classes, dateVector and appointmentTable, you need to define the javaOrganizer class. The javaOrganizer class is a public class that subclasses the Applet class because we are implementing this project as an applet. The class should contain the dateVector object, dates, as its member. It will be initialized in the init() method and will contain the user interface components as its members. Enter the following code into the file javaOrganizer.java, together with classes dateVector and appointment table.

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

public class javaOrganizer extends java.applet.Applet {

        dateVector dates;

        public void init() {
                dates = new dateVector();
        }

        TextArea apptEntry;
        TextArea apptView;
        TextField date;
        TextField hrs;
        TextField mins;
        Choice todo;
}
```

**6.** The start method in an applet is run at the start of an applet after initialization. So we will plug in an implementation of UI, into the start method, for this organizer. You can find details about the UI code in the chapters in this book which describes the Java AWT. Enter the following method into the class javaOrganizer.

```
public void start() {
                setLayout(new BorderLayout());

                Panel disp = new Panel();
                disp.setLayout(new FlowLayout());

                apptEntry = new TextArea(8,15);
                disp.add(apptEntry);
                apptView = new TextArea(8,15);
                apptView.setEditable(false);
                disp.add(apptView);
                add("Center", disp);

                Panel input_p = new Panel();
```

```
        input_p.setLayout(new FlowLayout());
        date = new TextField(2);
        input_p.add(date);
        hrs = new TextField(2);
        input_p.add(hrs);
        mins = new TextField(2);
        input_p.add(mins);

       todo = new Choice();
       todo.addItem("SetAppt");
       todo.addItem("FindAppt");
       input_p.add(todo);

       input_p.add(new Button("OK"));
       add("South", input_p);
       resize(600,200);

  }
```

**7.** Having provided the method to start the applet, the next step is to provide the necessary event handling. When a user clicks on the OK button, appropriate action is taken, depending on whether the option selected is to set the appointment or to find an appointment. The values for date, time, and details of an appointment are gathered from the user-interface. If the appointment is to be scheduled, then the setAppt method is called. This method adds an appointment to the appointmentTable object at the specified time and date. For this purpose, the appointmentTable for a specified date is retrieved and the new appointment is added. Enter the following code into the javaOrganizer class.

```
public boolean action(Event evt, Object arg) {
  if (arg.equals("OK")) {
    if (todo.getSelectedItem().equals("SetAppt")) {
       String time = hrs.getText() + mins.getText();
       String what = apptEntry.getText();
       int dat = Integer.parseInt(date.getText().trim());
       setAppt(dat, time, what);
    }
    if  (todo.getSelectedItem().equals("FindAppt")) {
       String time = hrs.getText() + mins.getText();
       int dat = Integer.parseInt(date.getText().trim());
       String what = findAppt(dat, time);
       apptView.setText("The appt on " + dat +" at "+
                    time +" is " + what);
     }
 }
 return true;
}

public  void setAppt(int date, String time, String what){
       appointmentTable appt = dates.getApptTab(date);
       appt.addOneAppt(time, what);
       dates.fixAppt(appt, date);
}
```

**8.** The next step is to define the method for finding the appointments for a specified date and time. This method returns a String object, which is then displayed in the view area in the UI. The dateVector object is indexed by the

specified date. By using the knowAppt method of the dateVector class, the appointment table for the specified date is obtained. Having obtained a handle to the dateVector for the specified date, the appointmentTable object for the specified time on that date is obtained by using the apptAt method of the appointmentTable class. Enter the following code inside the javaOrganizer class.

```
public String findAppt(int date, String time){
          String what = ((appointmentTable)dates.knowAppt(date)).
          apptAt(time);

   return what;
}
```

**9.** Having successfully completed the implementation of the javaOrganizer class, you can compile the class, using the javac compiler. Here is the HTML file to launch this applet. Enter it in a file named org.html.

```
<title>Java Appointment Organizer</title>
<hr>
<applet code=javaOrganizer.class width=250 height=250>
</applet>
<hr>
```

You can improve on this applet by adding methods to save appointments to a file and later retrieve the appointments from the file. Figure 12-15 shows the user-interface for this javaOrganizer applet.

**Figure 12-15** javaOrganizer: The Java Appointment Organizer in action

## How It Works

The applet is launched by either an applet viewer or from a browser. When the applet starts, it shows a user-interface as shown in Figure 12-15. There are three text fields at the bottom of the applet. The first field is for specifying the date, the second is for entering the hour, and the third field is for entering the minutes. Once the date and time are specified, you can either set an appointment for that time or find any existing appointment at that time. The choice button to the left of the OK button can be used to select your option. In the user-interface there are two text areas: one on the left and one on the right. The text area on the left is the area where you will enter the details of an appointment. If you want to set an appointment, choose the SetAppt option in the choice button and enter the details in the text area. Once you have completed the details, press the OK button to confirm the entry. If you want to find the appointment at the specified date and time, choose the FindAppt option in the choice button and press the OK button. The details of the appointment at the specified time are displayed in the text area on the

right side. You can keep track of your appointments by using this applet as long as you don't close the applet. You can provide the functionality to store the details to a file and retrieve it as needed, so that the applet can be closed at any time. You also need not worry about system crashes and power shutdowns if you extend this applet to provide that functionality. Good luck!

# Chapter 13
# Date And Advanced Classes

Chapter 12 described the utility classes for data structures and random number generation. In this chapter, you will learn more about the other utility classes provided in the JDK environment. We'll discuss the class that deals with sets of bits and the class that helps to tokenize a stream of strings, as well as a wrapper class for finding dates, and advanced classes that encapsulate the Observable-Observer design pattern.

The BitSet class represents sets of bits and provides operations on sets of bits, like logical OR, AND, and XOR. The StringTokenizer class is helpful in obtaining tokens from a string using the delimiter. Because Java is platform independent, manipulating the date independent of the system is very important and Java's Date class performs this function. The advanced utility class Observable provides the necessary behavior for objects to be observable by many other observer objects. When used with the Observer classes, this class helps design low-coupled, reusable systems that have a one-to-many dependency on changes that occur in observable classes. The project developed in this chapter displays the time using different clocks such as a digital clock and an analog clock. The clocks get the time from a central time server in a way that any change in time in the time server is observed and updated by every clock.

## BitSet

The BitSet class encapsulates a set of bits. The set can contain many bits and expands as more bits are added. It can be used to perform logical operations like AND, OR, and XOR between any two BitSets. Methods for clearing and setting bits are provided as member methods. You can clone a set of bits, because BitSet implements the Cloneable interface. This class is used to represent a set of boolean data. Using this class saves memory, as each value takes on only one bit for representation.

For example, consider two sets of bits A and B. There are four bits in each set. Let the values of the bits in A be 0, 0, 1, and 1. Let the values of the bits in set B be 0, 1, 0, and 1. Consider the logical AND, OR, and XOR operation over these two sets. In our discussion, a value of 1 indicates that the bit is set, and 0 indicates a clear bit. During an AND operation, the result of ANDing two bits is 1 if both the bits have a value of 1. If either of the bits is 1, the result is 0. In the case of an OR operation, the result value is 0 only if both the bits are of value 0. If one of them has a value of 1, the result is 1. If both

the bits have the same value and if you perform XOR operation on them, the result is 0. When you apply an XOR operation on two bits whose values vary, the result is 1. These relationships are expressed in Figure 13-1.

**Figure 13-1** Logical AND, OR, and XOR operations performed on sets of bits

## StringTokenizer

In computer languages, compilers play an important role. User-written code is parsed by the lexical analyzer to separate the individual tokens and verify their validity. In communications, messages are passed between entities and deciphered by the receiving party. In either environment, instead of parsing character by character, it is faster and more logical to parse tokens. A token can be a single word or a set of words. The delimiter specified in each case determines what constitutes a token. Delimiters are by default white space characters. Alternatively, you can specify your own delimiter, providing all the recipients are aware of it. This knowledge of the delimiter is vital in successful communications. Consider the string "Greetings. You are a Java Expert." If you consider white space as the delimiter, parsing the above string will result in six tokens, whereas, if you consider a period ('.') as the delimiter, there are only two tokens as shown in Figure 13-2. You can also specify whether the delimiters should be a part of the token. The StringTokenizer class in Java is useful to obtain tokens from a specified string.

**Figure 13-2** Tokens form the same string differ if the delimiter is different

## Date

In many applications you develop, obtaining the time and date or timing a process is important. You have to be aware of the system calls or library routines that provide these details and, most importantly, of how they differ from system to system, depending on the operating system. In Java, you are relieved of this trouble by the Date utility class, a wrapper class for dates. You can obtain the local time as well as the Greenwich Meridian Time (GMT), also known as Universal Time (UT) in technical terms. You can use the Date class to find out what day of the week you were born on, by specifying your date of birth. You can also use the Date class to time your processes and applications which will help you measure their performance. With the support of Java's AWT, you can generate graphs and bar charts too! This chapter covers methods available in the Date class and ways to use them in great detail.

## Observable-Observer

Imagine an application in which you want to define a one-to-many dependency between objects, so that when one of the objects experiences a change in its state, all its dependent objects are notified and updated automatically. This is a common design issue and the pattern it suggests is called Model-View, Observable-Observer, or Publish-Subscribe.

Consider the profits of a business firm. The firm concentrates on four major projects. You want to gather information about the percentage of profit each project contributes to the firm. Also you are interested in tabulating each profit. Instead of tightly coupling various information gathering programs, you can use one object to keep track of the profits. If it finds an observable change in the profit of the company, it notifies all the observer programs, which update their individual views as illustrated in Figure 13-3. Then if any observer needs more information, it can request it from the observable object (which is called the Subject in the Subject-Observer pattern). If the observable notifies the observers of all the changes, it is push-style programming. If the observable notifies the observers of a change and then they query for more details, it is pull-style programming.

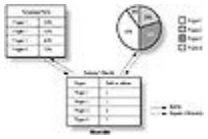

**Figure 13-3** Observable-Observer objects keep track of a firm's profits

This pattern is an interesting utility provided by the JDK. Observable is a class in Java, whereas, Observer is an interface. This interface has to be implemented by objects that want to be one of the observers. Details of using the Observable class and Observer interface are presented with examples in this chapter.

## Date and Advanced Classes Summaries

Table 13-1 summarizes the classes and interface for the BitSet, StringTokenizer, Observable, Observer, and Date utilities.

**Table 13-1** Summary of Utility Classes

| Class/Interface Name | Description |
|---|---|
| BitSet | Represents a set of bits and expands automatically as more bits are added to the set. |
| StringTokenizer | Encapsulates the linear tokenization of a String. |
| Observable | The Subject class in a Subject-Observer pattern and can have many observers. |
| Observer | An Interface, useful for objects to act as observers of an |

| | |
|---|---|
| | observable object and keep track of the changes to it. |
| Date | A wrapper class for dates, useful for manipulating dates independent of the system. |

## BitSet

**Purpose**

Represents a set of bits and expands automatically as more bits are added to the set.

**Syntax**

public class BitSet extends Object implements Cloneable

**Description**

BitSet represents a set of bits. The set size increases as more bits are introduced. It is useful to logically operate on a set of bits. This class implements the Cloneable interface and should define the clone() method, so it can be cloned. BitSet supports operations like AND, OR, and XOR of the target set with another set. Figure 13-4 illustrates the inheritance relationship of class BitSet.

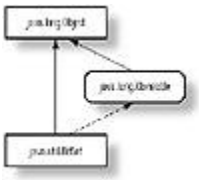

**Figure 13-4** Class diagram of the BitSet class

**PackageName**

*java.util*

**Imports**

*import java.util.BitSet;*

**Constructors**

public BitSet()
public BitSet(int *size*)

**Parameters**

***size***

The initial size of the set of bits to be created.

**Example**

The bitsetDemo class in Listing 13-1 illustrates the use of the BitSet class. It has a member of type BitSet and is used to demonstrate the methods of this class.

**Listing 13-1** bitsetDemo.java: Code demonstrating the usage of Bitset class and its methods

```
import java.util.BitSet;

public class bitsetDemo {
```

```
BitSet bs;

public bitsetDemo() {
       bs = new BitSet();

}

public bitsetDemo(int size) {
       bs = new BitSet(size);

}

public void debug(String str) {
       System.out.println(str);

}

public void setStatus(int bitno) {
       bs.set(bitno);

}

public boolean getStatus(int bitno) {
       return bs.get(bitno);

}

public BitSet allStatus() {
     debug(" Cloning bit set of size " + bs.size());
       return (BitSet)bs.clone();

}

public BitSet andWith(BitSet tmp) {
       if (tmp.equals(bs))

return tmp;

       debug(" When " + tmp + " is ANDed with " + bs +
        " result is ");
       tmp.and(bs);
       debug(" " + tmp + "\n");
       return tmp;
}

public BitSet orWith(BitSet tmp) {
       if (tmp.equals(bs))
             return tmp;
       debug(" When " + tmp + " is ORed with " + bs + " result
       is ");
       tmp.or(bs);

       debug(" " + tmp + "\n");
       return tmp;
```

```
        }

        public void clearAll() {
               debug(" hashCode of the BitSet is " +   bs.hashCode());
               bs.xor(bs);
               debug(" After clearAll --> " + bs.toString());

        }

        public final static void main(String args[]) {
               bitsetDemo bd = new bitsetDemo(5);
               bd.setStatus(0);
               bd.setStatus(4);
               System.out.println(bd.getStatus(0) +
               " , " + bd.getStatus(1));
               bd.setStatus(6);
               BitSet newset = bd.allStatus();
               System.out.println(newset);
               newset.clear(4);
               bd.andWith(newset);
               bd.orWith(newset);
               newset.set(4);
               bd.clearAll();
        }
}
```

## and(BitSet)

**ClassName**

BitSet

**Purpose**

Logically ANDs the target object with the specified object.

**Syntax**

public void and(BitSet bs)

**Parameters**

***bs***

The specified BitSet object that the target object is ANDed.

**Description**

This method applies the logical AND operation to the bits in the target object and the bits in the specified object, *bs*. Each bit in the target object is ANDed with the bits at the corresponding index in the specified bit set. The resultant object is the modified target object.

**Imports**

*import java.utils.BitSet;*

**Returns**

None.

**See Also**

The or(BitSet) method in theBitSet class

**Example**

Refer to the method andWith in Listing 13-1. It ANDs two bit sets by using this method.

## clear(int)

**ClassName**
BitSet
**Purpose**
Clears the specified bit.
**Syntax**
public void clear(int index)
**Parameters**
***index***
The index of the bit in the bit set that has to be cleared, that is set to 0.
**Description**
The bit at the specified index is cleared by using this method. This way you can selectively clear any individual member in the bit set.
**Imports**
*import java.utils.BitSet;*
**Returns**
None.
**See Also**
The set method in the BitSet class
**Example**
Refer to the main method in Listing 13-1. The new_set object has its bit at index 4 cleared using this method.

## clone()

**ClassName**
BitSet
**Purpose**
This method clones the bit set.
**Syntax**
public void clone()
**Parameters**
None.
**Description**
A clone of the BitSet is created using this method. It doesn't copy the values into the new bit set, but creates a shallow copy. This method overrides the clone method in class Object.
**Imports**
*import java.utils.BitSet;*
**Returns**
The clone of the target BitSet object. Return type is Object, which you may typecast to BitSet.
**See Also**
The Cloneable interface
**Example**

Refer to the allStatus method in Listing 13-1. The Bitset is cloned and the newly created clone is returned by the method.

## equals(Object)

**ClassName**

BitSet

**Purpose**

Returns a boolean value indicating if the target object is equal to the specified bit set object. It returns true if the objects are the same and false if they are not equal.

**Syntax**

public void equals(Object obj)

**Parameters**

***obj***

The object instance to be compared with.

**Description**

The bit at the specified index is cleared using this method. This way you can selectively clear any individual member in the bit set.

**Imports**

*import java.utils.BitSet;*

**Returns**

None.

**Example**

Refer to the main method in Listing 13-1. The new_set object has its bit at index 4 cleared using this method.

## get(int)

**ClassName**

BitSet

**Purpose**

Obtains the specified bit.

**Syntax**

public boolean get(int index)

**Parameters**

***index***

The index of the bit in the bit set that has to be obtained.

**Description**

The bit at the specified index is retrieved using this method.

**Imports**

*import java.utils.BitSet;*

**Returns**

A boolean value indicating the value of the bit at the specified location.

**See Also**

The set method in the BitSet class.

**Example**

Refer to the getStatus method in Listing 13-1. This method is used to obtain the bit at the specified location.

## hashCode()

**ClassName**

BitSet

**Purpose**

Obtains the hash code of the target bit set object.

**Syntax**

public int hashCode()

**Parameters**

None.

**Description**

The hash code of the target bit set object is returned by this method. It overrides the hashCode method of class Object.

**Imports**

*import java.utils.BitSet;*

**Returns**

The hash code of the target object. Return type is int.

**Example**

Refer to the clearAll method in Listing 13-1. The hashCode method is used to debug the object information.

## or(BitSet)

**ClassName**

BitSet

**Purpose**

Logically ORs the target object with the specified object.

**Syntax**

public void or(BitSet bs)

**Parameters**

***bs***

The specified BitSet object that the target object is ORed.

**Description**

This method applies the logical OR operation to the bits in the target object and the bits in the specified object, *bs*. Each bit in the target object is ORed with the bits at the corresponding index in the specified bit set. The resultant object is the modified target object.

**Imports**

*import java.utils.BitSet;*

**Returns**

None.

**See Also**

The and(BitSet) method in the BitSet class

**Example**

Refer to the orWith method in Listing 13-1. It ORs two bit sets by using this method.

## set(int)

**ClassName**

BitSet

**Purpose**

Sets the specified bit.

**Syntax**

public void set(int index)

**Parameters**

***index***

The index of the bit that has to be set.

**Description**

The bit at the specified index is set using this method. The value of the bit will be made 1.

**Imports**

*import java.utils.BitSet;*

**Returns**

None.

**See Also**

The get method in the BitSet class.

**Example**

Refer to the setStatus method in Listing 13-1. This method is used to set the value of the bit at the specified location.

## size()

**ClassName**

BitSet

**Purpose**

Obtains the size of the bit set.

**Syntax**

public int size()

**Parameters**

None.

**Description**

The size of the bit set is obtained by using this method. This value reflects the number of bits in the set.

**Imports**

*import java.utils.BitSet;*

**Returns**

A value indicating the size of the bit set is returned. Return type is int.

**Example**

Refer to the allStatus method in Listing 13-1 where the number of bits in the set is printed to the screen.

## toString()

**ClassName**

BitSet

**Purpose**

Obtains the string form of the bit set.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

The string representation of the target BitSet object is returned by this method. This method will return a string containing the index of the bits that are set in the bit set. This is usually used for debugging purposes.

**Imports**

*import java.utils.BitSet;*

**Returns**

A string representing the details of the bit set; return type is String.

**Example**

Refer to the clearAll method in Listing 13-1. This toString method is used to obtain the string representation of the bit set.

## xor(BitSet)

**ClassName**

BitSet

**Purpose**

Logically XORs the target object with the specified object.

**Syntax**

public void xor(BitSet bs)

**Parameters**

***bs***

The specified BitSet object that the target object is XORed.

**Description**

This method applies the logical XOR operation to the bits in the target object and the bits in the specified object, *bs*. Each bit in the target object is XORed with the bits at the corresponding index in the specified bit set. The resultant object is the modified target object. When two identical objects are XORed, the result is an object with all bits cleared.

**Imports**

*import java.utils.BitSet;*

**Returns**

None.

**See Also**

The and() and or() methods in the BitSet class

**Example**

Refer to the clearAll method in Listing 13-1. All the bits in the bit set are cleared by the single use of this method on the same object, *bs*.

## StringTokenizer

**Purpose**

Encapsulates the linear tokenization of a String.

**Syntax**

public class StringTokenizer extends Object implements Enumeration

**Description**

StringTokenizer controls the linear tokenizing of strings. It is helpful for parsing strings and to get tokens out of a string. When messages are exchanged between two objects, instead of sending strings for every piece of information, it is useful to send a single string that contains multiple pieces of information, each separated by a delimiter. You can use an instance of StringTokenizer to tokenize the individual pieces of information from the message String received. This class implements the Enumeration interface and, hence, defines the hasMoreElements and nextElement methods of that interface. Figure 13-5 illustrates the inheritance relationship of the StringTokenizer class.

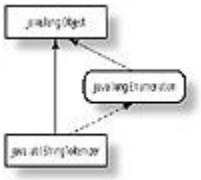

**Figure 13-5** Class diagram of the StringTokenizer class

**PackageName**

*java.util*

**Imports**

*import java.util.StringTokenizer;*

**Constructors**

public StringTokenizer(String *str*)
public StringTokenizer(String str, String *delim*)
public StringTokenizer(String *str*, String *delim*, boolean *returnDelimsToo*)

**Parameters**

***str***

The string to be tokenized.

***delim***

The delimiter specified for use in tokenizing.

***returnDelimsToo***

A boolean value indicating whether or not you want the delimiters to be returned as tokens.

**Example**

The docGenerator class in Listing 13-2 demonstrates the use of the StringTokenizer class and its methods.

**Listing 13-2** docGenerator.java: Demonstrating the usage of StringTokenizer and its methods

```
import java.util.StringTokenizer;

public class docGenerator extends StringTokenizer {

       String message;
       String delimiter;

       public docGenerator(String msg) {
               super(msg, ":", false);
               message = msg;
       }

       public docGenerator(String msg, String delim) {
              super(msg, delim, false); // skip the delimiter tokens
              message = msg;
              delimiter = delim;
      }

      public void decipherMesg() {
              int num_tokens = countTokens();
              if (num_tokens-- > 0) {
               System.out.print(" The book title is --> ");
               System.out.println(nextElement());

              }

              if (num_tokens-- > 0) {
               System.out.print(" Publisher --> ");
               System.out.println(nextToken());
              }

              if (num_tokens-- > 0) {
               String authors = nextToken();
               System.out.println(" The authors are ");
               docGenerator new_dg = new docGenerator(authors);
               while (new_dg.hasMoreTokens())
                       System.out.println("  " +new_dg.nextToken(" "));
              }
              //if (num_tokens-- > 0 ) {

      }

      public final static void main(String args[]) {
              String s = "Java SuperBible:Waite Group:Nataraj Brian
Arvind";
              docGenerator dg = new docGenerator(s);
```

```
            dg.decipherMesg();
        }
}
```

## countTokens()

**ClassName**
StringTokenizer
**Purpose**
Obtains the number of tokens in the string using the specified delimiter.
**Syntax**
public int countTokens()
**Parameters**
None.
**Description**
This method counts the number of tokens in the string that is to be tokenized. The current delimiter is used as the delimiter while counting the tokens. This method returns the number of tokens in the string. The value returned is the number of times the nextToken() method is to be called until the end of the stream is reached.
**Imports**
*import java.utils.StringTokenizer;*
**Returns**
The number of tokens in the string using the current delimiter set. Return type is int.
**See Also**
The nextToken method in the StringTokenizer class
**Example**
Refer to the decipherMesg method in Listing 13-2. The number of tokens is found using the countTokens method and it is used to decipher each token.

## hasMoreElements()

**ClassName**
StringTokenizer
**Purpose**
Returns a boolean value indicating whether or not the enumeration has more elements.
**Syntax**
public boolean hasMoreElements()
**Parameters**
None.
**Description**
This method uses the current delimiter and enumerates the tokens as elements in the string. It returns true if there are more elements in the enumeration of the tokens in the string using the current delimiter. If there are no more elements in

the enumeration, this method returns false. This method is defined because the StringTokenizer class implements the Enumeration interface.

**Imports**

*import java.utils.StringTokenizer;*

**Returns**

A value of true or false, depending on whether or not there are more elements in the enumeration of the string. Return type is boolean.

**See Also**

The hasMoreTokens method in the StringTokenizer class; the Enumeration interface

**Example**

Refer to the decipherMsg method in Listing 13-2. The tokens of the string containing the authors' names with a delimiter of whitespace " " are enumerated using the hasMoreTokens method. Even if you use the hasMoreElements method instead, the effect is the same.

## hasMoreTokens()

**ClassName**

StringTokenizer

**Purpose**

Returns a boolean value indicating whether or not there are more tokens in the string using the current delimiter.

**Syntax**

public boolean hasMoreTokens()

**Parameters**

None.

**Description**

This method uses the current delimiter and obtains all the tokens in the string. It returns true if there are more tokens to be tokenized in the string using the current delimiter. If there are no more tokens in the string, this method returns false.

**Imports**

*import java.utils.StringTokenizer;*

**Returns**

A value of true or false depending on whether or not there are more tokens in the string. Return type is boolean.

**See Also**

The hasMoreElements method in the StringTokenizer class

**Example**

Refer to the decipherMsg method in Listing 13-2. The tokens of the string containing the authors' names with a delimiter of whitespace " " are obtained one by one using the hasMoreTokens method.

## nextElement()

**ClassName**

StringTokenizer

**Purpose**

Obtains the next object in the enumeration of the tokens in a string.

**Syntax**

public Object nextElement()

**Parameters**

None.

**Description**

This method uses the current delimiter and enumerates the tokens as elements in the string. After enumerating the tokens and making successive calls to this method, the successive elements are returned. It is defined because the StringTokenizer class implements the Enumeration interface. If there are no elements in the enumeration, it throws a NoSuchElementException. This exception should be caught in a try-catch block. It is advisable to use this method only if the hasMoreElements method returns true.

**Imports**

*import java.utils.StringTokenizer;*

**Returns**

The next element in the enumeration, which is the next token in the string using the current delimiter. Return type is Object, which can be typecast to String.

**See Also**

The nextToken method in the StringTokenizer class

**Example**

Refer to the decipherMesg method in Listing 13-2. The book title is obtained by getting the first element in the token set using this method. It would have the same effect if you replace nextElement with the call to the nextToken method.

## nextToken()

**ClassName**

StringTokenizer

**Purpose**

Obtains the next string token from the enumeration of the tokens in a string.

**Syntax**

public String nextToken()

**Parameters**

None.

**Description**

This method uses the current delimiter and enumerates the tokens as elements in the string. After enumerating the tokens, successive calls to this method returns successive elements. It throws a NoSuchElementException if there are no tokens in the String. This exception should be caught in a try-catch block. It is advisable to use this only if the hasMoreTokens method returns true.

**Imports**

*import java.utils.StringTokenizer;*

**Returns**

The next token in the string using the current delimiter is returned. Return type is String.

**See Also**

The nextElement method in the StringTokenizer class

**Example**

Refer to the decipherMesg method in Listing 13-2. The publisher's name is obtained using this method. It would have the same effect if you replaced nextToken with a call to the nextElement method.

## nextToken(String)

**ClassName**

StringTokenizer

**Purpose**

Obtains the next string token from the enumeration of the tokens in a string using the specified string as delimiter.

**Syntax**

public String nextToken(String delim)

**Parameters**

***delim***

The delimiter to be used when tokenizing the string.

**Description**

This method uses the specified delimiter and enumerates the remaining tokens as elements in the string. After enumerating the tokens, successive calls to this method return successive elements in the enumeration. The default delimiter for strings if you don't specify one during construction is " \n\t\r", the whitespace characters. You can switch delimiters using this method to get successive tokens. If this method is used, the specified delimiter will remain as the default delimiter for the rest of the tokenizing session, until it is changed again.

**Imports**

*import java.utils.StringTokenizer;*

**Returns**

The next token in the string using the specified delimiter; return type is String.

**See Also**

The nextElement and nextString methods in the StringTokenizer class

**Example**

Refer to the decipherMesg method in Listing 13-2. The authors' names are obtained using this method. The names are separated by white spaces, while the categories (title, publisher name, authors' names) are separated by ":" as delimiter. Switching the delimiter to a single space is done using this method.

## Observable

**Purpose**

The Subject class in a Subject-Observer pattern, which can have many observers.

**Syntax**

public class Observable extends Object

**Description**

This class should be subclassed by an object that is observable by other observers. This class forms the Subject class in a Subject-Observer pattern or the "data" in the Model-View paradigm. It defines a one-to-many dependency between objects so that when this Observable object changes the observer objects are notified. Notification is achieved by calling the update method of all the observers of this observable object. This design strategy helps in low coupling between the objects. It has methods to add and delete observers and to notify them. Figure 13-6 illustrates the inheritance relationship of class Observable.

**Figure 13-6** Class diagram of the Observable class

**PackageName**

*java.util*

**Imports**

*import java.util.Observable;*

**Constructors**

public Observable()

**Parameters**

None.

**Example**

The clockTimer class in Listing 13-3 subclasses the Observable class. During clock ticks, it notifies all its observer objects, which are instances of digitalClock in this case.

**Listing 13-3** clockTimer.java: Illustrates the usage of Observable class and Observer interface

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

public class clockTimer extends Observable implements Runnable {

        int hour;
        int mins;
        int secs;
        Date date;

          public clockTimer() {
```

```
            date = new Date();
            hour = date.getHours();
            mins = date.getMinutes();
            secs = date.getSeconds();
            start();
        }

        Thread timeThread;

        public void start() {
                if (timeThread == null) {
                        timeThread = new Thread(this);
                        timeThread.start();
                }
        }

        public int GetHours() {
                return hour;
        }

        public int GetMinutes() {
                return mins;
        }

        public int GetSeconds() {
                return secs;
        }

        public void run() {
             while (timeThread != null) {
                 try {
Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
                        tick();
                 Thread.sleep(1000);
                  } catch (InterruptedException e){
debug("exception");}
            }
        }

        public void debug(String str) {
                System.out.println(str);
        }

        public synchronized void tick() {

                date = null;
                clearChanged();
                Date new_date = new Date();
                if (new_date.getSeconds() != secs) {
                        setChanged();
                        date = new_date;
                        hour = date.getHours();
                        mins = date.getMinutes();
                        secs = date.getSeconds();
                }
                notifyObservers();
        }
```

```
        public final static void main(String args[]) {
                clockTimer ct = new clockTimer();
                ct.start();
        }

}

class DigitalClock implements Observer, Runnable {

        clockTimer myTimer;
        int hours;
        int mins;
        int secs;

        public DigitalClock(clockTimer timer) {
             myTimer = timer;
             hours = myTimer.GetHours();
             mins = myTimer.GetMinutes();
             secs = myTimer.GetSeconds();
             myTimer.addObserver(this);
             start();
        }
        Thread clockThread;

        public void start() {
             if (clockThread == null) {
                     clockThread = new Thread(this);
                  clockThread.start();
             }
        }

        public void run() {

              while (clockThread != null) {
                try {
                    clockThread.sleep(1000);
                    } catch (InterruptedException e){}
              }

        }

        public void update(Observable obs, Object arg) {
              if (obs.hasChanged()) {
                   myTimer = null;
                        myTimer = (clockTimer)obs;
                   hours = myTimer.GetHours();
                   mins = myTimer.GetMinutes();
                   secs = myTimer.GetSeconds();
              }
                display();
        }

        private void debug(String str) {
              System.out.println(str);
        }
```

```
        private void display() {
                debug("Notified by observer");
        }

}
```

## addObserver(Observer)

**ClassName**
Observable
**Purpose**
Adds the specified observer to the observer list for this observable object.
**Syntax**
public synchronized void addObserver(Observer obs)
**Parameters**
***obs***
The observer object for this observable to be added to the list.
**Description**
This method adds the specified observer object to the observers list. When a change occurs in the observable object all its observers are notified, so the specified object will be notified of changes in the observable object.
**Imports**
*import java.utils.Observable;*
**Returns**
None.
**See Also**
The deleteObserver and deleteObservers methods in the Observable class
**Example**
Refer to Listing 13-3. The digitalClock object on construction adds itself to the observer list using this method on the clockTimer object.

## clearChanged()

**ClassName**
Observable
**Purpose**
Clears the flag within the observable object indicating no change in observable object.
**Syntax**
public synchronized void clearChanged()
**Parameters**
None.
**Description**
This method clears any change in the target observable object. No observer will be notified before a change occurs and the hasChanged method returns true.

**Imports**

*import java.utils.Observable;*

**Returns**

None.

**See Also**

The setChanged and hasChanged methods in the Observable class

**Example**

Refer to Listing 13-3. In the tick()method, in the clockTimer class, before any change is noted in the observable time (any change in seconds), this method is used to clear the flag that indicates a change in the observable object.

## countObservers()

**ClassName**

Observable

**Purpose**

Obtains the number of observers in the observers list of this observable object.

**Syntax**

public synchronized int countObservers()

**Parameters**

None.

**Description**

This method counts and returns the number of observer objects that are observing this observable object. When a change occurs in the observable object all its observers are notified.

**Imports**

*import java.utils.Observable;*

**Returns**

The count of the number of observer objects in the observer list of the observable object. Return type is int.

**Example**

Refer to Listing 13-3. You can use this method in the Observable clockTimer class to find out the number of observers.

## deleteObserver(Observer)

**ClassName**

Observable

**Purpose**

Removes the specified observer from the observer list for this observable object.

**Syntax**

public synchronized void deleteObserver(Observer obs)

**Parameters**

***obs***

The observer object, for this observable object to be deleted from the list.

**Description**

This method deletes the specified observer object from the observer list. The specified object will not be notified of changes to the observable object.

**Imports**

*import java.utils.Observable;*

**Returns**

None.

**See Also**

The addObserver and deleteObservers method in the Observable class

**Example**

Refer to Listing 13-3. You can delete an observer by calling this method. The observers add themselves by calling addObserver and, when desired, they can delete themselves from the list using this method.

## deleteObservers()

**ClassName**

Observable

**Purpose**

Removes all the observers from the observer list for this observable object.

**Syntax**

public synchronized void deleteObservers()

**Parameters**

None.

**Description**

This method deletes all the observer objects from the observer list. There will not be any observer object to be notified of any change that happens to the observable object.

**Imports**

*import java.utils.Observable;*

**Returns**

None.

**Example**

Refer to Listing 13-3. The clockTimer object can delete all observers after a couple of ticks, and this will result in no update of time in the observer class.

## hasChanged()

**ClassName**

Observable

**Purpose**

Returns a boolean value, indicating whether or not there is any change in the observable object.

**Syntax**

public synchronized boolean hasChanged()

**Parameters**

None.

**Description**

This method returns a boolean value indicating whether or not the target observable object has undergone a change. This method is used to decide whether the observers should be notified of any change. This method returns true if there is any change. If there is no change in the observable object, this method returns false.

**Imports**

*import java.utils.Observable;*

**Returns**

Returns true if there is a change in the observable object; false otherwise. Return type is boolean.

**See Also**

The clearChanged method in the Observable class

**Example**

Refer to Listing 13-3. In the update method in the digitalClock class, before updating the new values, the observable object is checked to see if there is any change in the object. If there is, then the values of the observers are changed and the display method is called.

## notifyObservers()

**ClassName**

Observable

**Purpose**

Notifies all the observers of the observable object of a change in the object.

**Syntax**

public synchronized void notifyObservers()

**Parameters**

None.

**Description**

This method notifies all the observers in the list of observers in the target Observable object when a change occurs. It results in a call to the update method of the observer objects in the list.

**Imports**

*import java.utils.Observable;*

**Returns**

None.

**See Also**

The notifyObservers(Object) method in the Observable class

**Example**

Refer to the tick method in the clockTimer class in Listing 13-3. After there is a change in the time of the clockTimer objects, all the observers (in this case, digitalClock objects) are notified of the change.

## notifyObservers(Object)

**ClassName**

Observable

**Purpose**

All the observers of the observable object are notified of the change in the specified object in the target Observable object.

**Syntax**

public synchronized void notifyObservers(Object arg)

**Parameters**

***arg***

The changed object whose change has to be made known to observer objects.

**Description**

This method notifies all the observers in the list of observers in the target Observable object of the change in the specified Object arg. This method is used to notify the observers when a change occurs, and results in a call to the update method of the observer objects in the list. The changed variable arg is also passed as a reference to the update method of the observer objects. This is helpful in that each object checks this arg parameter for the change the observer is interested in. The observers need not check for all the changes that have occurred in the target observable object.

**Imports**

*import java.utils.Observable;*

**Returns**

None.

**See Also**

The notifyObservers() method in the Observable class

**Example**

Refer to the tick method in class clockTimer in Listing 13-3. After a change in the time of the clockTimer objects, all the observers (in this case, digitalClock objects) are notified of changes. But if you replace this method with the notifyObservers(arg) method by passing the date object, then the observer can check the changed date. There can be observers that are not observing the change in time and can safely ignore the notification of change in time.

## setChanged()

**ClassName**

Observable

**Purpose**

Sets the flag within the observable object indicating a change in the observable.

**Syntax**

public synchronized void setChanged()

**Parameters**

None.

**Description**

This method sets a flag indicating a change in the target observable object. An Observer can be notified of this change by calling the notifyObservers method.

**Imports**

*import java.utils.Observable;*

**Returns**

None.

**See Also**

The clearChanged and hasChanged methods in the Observable class

**Example**

Refer to Listing 13-3. In the tick() method, in class clockTimer, after the change in the observable time (any change in seconds) is noted, this method is used to set the flag that indicates a change in the observable object.

## Observer

**Purpose**

An Interface useful for objects to act as observers of an observable object and keep track of any changes to the observable object.

**Syntax**

public interface Observer extends Object

**Description**

This interface should be implemented by a class that has to behave as an observer of changes in an observable object. This class forms the Observer class in a Subject-Observer pattern or the "viewer" in the Model-View paradigm. An object can behave as an observer for many observable objects. It lists its interest in observing an observable by adding itself to the observers list of the observable object. Any change in the observable object will be communicated to the class implementing this interface by using the update method in this interface. Any class that implements this interface should define the update class such that when notified by the observable object, the target observer object takes appropriate action on the change that has occurred. This design strategy helps in low coupling between the objects. Figure 13-7 illustrates the inheritance relationship of interface Observer.

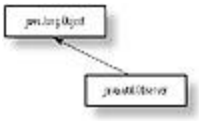

**Figure 13-7** Class diagram of the Observer interface

**PackageName**

*java.util*

**Imports**

*import java.util.Observer;*

**Example**

The DigitalClock class in Listing 13-3 implements this interface Observer. Any change in the observable clockTimer class is communicated to this DigitalClock

object, so that the DigitalClock object can take appropriate action by using its update method.

## update(Observable, Object)

**ClassName**

Observer

**Purpose**

Method called by the observable when it notifies all the observers.

**Syntax**

public abstract void update(Observable obs, Object arg)

**Parameters**

***obs***

The observable that calls this update method.

***arg***

The object that has changed in the observable object.

**Description**

This method is called by the observable object on the target Observer object. The target object is an observer in the list of observers in the observable object. The parameter *obs*, is the observable that notifies of the change. This parameter can be used to distinguish between notification calls from different observables, when the target observer object is in the list of many observable objects. This method is called on the observer object when any of the observable objects change and call the notifyObservers method. This is an abstract method and should be defined in the class that implements the Observer interface.

**Imports**

*import java.utils.Observer;*

**Returns**

None.

**See Also**

The notifyObservers method in the Observable class

**Example**

Refer to Listing 13-3. The update method is defined in the DigitalClock class, which implements the Observer interface. In this method the change in time is observed.

## Date

**Purpose**

A wrapper class for date, which is useful to manipulate dates in a system-independent way.

**Syntax**

public class Date extends Object

**Description**

This is a wrapper object for a date that contains details of date, month, year, hour, minutes, and seconds. It depends on the local time zone. The standard for time used in computers is the Greenwich Meridian Time (GMT), which is equivalent to the Universal Time (UT). The behavior is not 100 percent accurate as it does not reflect the addition of the 'leap second,' introduced every year. The time depends on the underlying operating system and its accurate maintenance of time. For construction, you can specify a set of values for the Date to be created. The arguments need not necessarily fall within the specified range. The fields are normalized before the Date object is created. For example, the 32nd of March will be interpreted as 1st of April. Figure 13-8 illustrates the inheritance relationship of class Date.

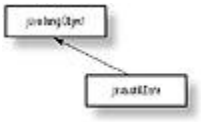

**Figure 13-8** Class diagram of the Date class

**PackageName**

*java.util*

**Imports**

*import java.util.Date;*

**Constructors**

public Date()
public Date(long *arg*)
public Date(int *year*, int *month*, int *date*)
public Date(int *year*, int *month*, int *date*, int *hours*, int *mins*)
public Date(int *year*, int *month*, int *date*, int *hours*, int *mins*, int *secs*)
public Date(String *str*)

**Parameters**

***arg***

Time in milliseconds corresponding to the date to be created.

***year***

A year after 1900.

***month***

A month number between 0-11.

***date***

A day of a month with a value between 1-31.

***hours***

An hour of the day between 0-23.

***mins***

The minutes past an hour, with a value between 0-59.

***secs***

The seconds past the minute, with a value between 0-59.

**Example**

The dateDemo class in Listing 13-4 demonstrates the usage of methods of the Date class. The member of the class is date of type Date. Methods of this object are invoked by using the methods of the dateDemo class.

**Listing 13-4** dateDemo.java: Demonstrates the usage of methods of Date class

```
import java.util.Date;

class dateDemo {

      Date date;

      public dateDemo() {
           date = new Date();
      }

      public void debug(String str) {
           System.out.println(str);
      }

      public String whatDay(int day) {
           switch (day) {
            case 0: return "Sunday";
            case 1: return "Monday";
            case 2: return "Tuesday";
            case 3: return "Wednesday";
            case 4: return "Thursday";
            case 5: return "Friday";
            case 6: return "Saturday";
            default: return "Oops!Wrong day";
           }
      }

      public void today() {

           debug("Today is " + whatDay(date.getDay())+ " " +
           date.getMonth() + "/"+date.getDate() +
                 "/19" + date.getYear());
      }
      public void newYear(int yr) {
           int day = (new Date(yr, 0, 1)).getDay();
           debug(" The new year's day for the year 19"+ yr
           + " fell on
           a " + whatDay(day));
      }

      public void now() {
           int secs = date.getSeconds();
           String sec_str;
           if (secs < 10)
                  sec_str = "0"+secs;
           else
                  sec_str = ""+secs;
           debug("Now the time is " + date.getHours() + ":" +
           date.getMinutes()+ ":" + sec_str);
```

```
            debug(" Hash code for the date representing this moment is
            "+ date.hashCode());
            debug(" Present time in ");
            debug("      GMT convention is     " + date.toGMTString());
            debug("      Locale conventions is " +
            date.toLocaleString());

            debug(" The time zone offset is " + date.getTimezoneOffset()
            + " minutes");
      }
      public void timeTheMethod() {
            Date start = new Date();
            for (int i=0;i<10000;i++);
            Date end = new Date();
            String s = " Time taken for making 10000 loops is ";
            long timing = (end.getTime() - start.getTime());
            s += "" + timing + " milli secs";
            debug(s);
            if (!start.equals(end))
                  debug(" Even if dates differ by milliseconds, they
are
            considered different object
                        instances of Date class");
            if (start.before(end))
                  debug(" And earlier created date object represents
time
            before the later created
                        object ..ofcourse, no surprise");
         }

      public void getTimeFromString(String str) {
            debug(" Time value of " + str + " is " + Date.parse(str));
      }

      public void demoUTCtimes() {
            Date  d = new Date();
            // "Thu, 22 Feb 1996 12:00:00 GMT"
            d.setDate(22);
            d.setMonth(1);
            d.setYear(96);
            d.setHours(12);
            d.setMinutes(00);
            d.setSeconds(00);
            long time = Date.UTC(96,1,22,12,00,00);
            d.setTime(time);
            if (time == d.getTime())
                        debug(" Time obtained by using UTC method and by
                        explicitly setting all parameters of
                                    the date object are the same");

      }

      public final static void main(String args[]){
            dateDemo d_demo = new dateDemo();
            d_demo.today();
            d_demo.newYear(73);
            d_demo.now();
```

```
            Date d1 = new Date();

            d1.setDate(21);
            d1.setMonth(1);
            d1.setYear(96);
            String str = " Details for 21/2/96:  ";
            str += d1.toString();
            System.out.println(str);

            d_demo.timeTheMethod();
            d_demo.getTimeFromString("Thu, 22 Feb 1996 12:00:00 GMT");
            d_demo.demoUTCtimes();

      }
}
```

## UTC(int, int, int, int, int, int)

**ClassName**

Date

**Purpose**

Method that calculates the Coordinated Universal Time (UTC) from the specified year, month, date, hours, minutes, and seconds values passed as parameters to this method. The parameters are not interpreted in the local time zone, but in UTC. This is a static method and hence can be referenced using the class name.

**Syntax**

public static long UTC(int year, int month, int date, int hours, int mins, int secs)

**Parameters**

***year***

A year after 1900.

***month***

Month of the year, with values between 0-11.

***date***

Day of the month, with values between 1-31.

***hours***

Hour of the day, with values between 0-23.

***mins***

Minutes past an hour, with values between 0-59.

***secs***

Seconds past the minute, with values between 0-59.

**Description**

This method is used to calculate a UTC value from the given parameters. The method interprets the parameters in UTC and not in the local time zone.

**Imports**

*import java.utils.Date;*

**Returns**

Returns the time value; return type is long.

**Example**

Refer to the demoUTCtimes method in Listing 13-4. This UTC method is used for finding the UTC time.

## after(Date)

**ClassName**

Date

**Purpose**

Returns a boolean value indicating whether or not the target date comes after the specified date.

**Syntax**

public boolean after(Date date)

**Parameters**

***date***

An instance of the Date class.

**Description**

A date is considered to come after another date, if it occurs after the other date in chronological order. This method returns true if the date value of the target object comes after the value of the specified date object; otherwise, this method returns false.

**Imports**

*import java.utils.Date;*

**Returns**

A boolean value indicating whether the target object came earlier or later than the specified object.

**See Also**

The before(date) method in the Date class

**Example**

Refer to Listing 13-4. In the method named timeTheMethod, this method is used to determine whether a date falls after a specified date.

## before(Date)

**ClassName**

Date

**Purpose**

Returns a boolean value indicating whether or not the target date comes before the specified date.

**Syntax**

public boolean before(Date date)

**Parameters**

***date***

An instance of the Date object.

**Description**

A date is considered to come before another date, if it occurs before the other date in chronological order. This method returns true if the date value of the target object comes before the value of the specified date object; otherwise, this method returns false.

**Imports**

*import java.utils.Date;*

**Returns**

A boolean value indicating whether the target object came earlier than the specified object.

**See Also**

The after method in the Date class

**Example**

Refer to Listing 13-4. In the method named timeTheMethod, the after method is used to determine whether a date falls after a specified date. You can use the before() method in a similar manner.

## equals(Object)

**ClassName**

Date

**Purpose**

Returns a boolean value indicating whether or not the target date equals the specified date.

**Syntax**

public boolean equals(Object date)

**Parameters**

***date***

An instance of the Date object.

**Description**

A Date object is equal to another Date object if they have the same values of date and time. This method returns true if the target date equals the date object; otherwise, this method returns false.

**Imports**

*import java.utils.Date;*

**Returns**

A boolean value indicating whether or not the target object is equal to the specified object.

**Example**

Refer to Listing 13-4. In the timeTheMethod method execution, the date start and date end are checked for equality.

## getDate()

**ClassName**

Date

**Purpose**

Returns the day of the month in a Date object.

**Syntax**

public int getDate()

**Parameters**

None.

**Description**

This method returns the day of the month. The returned value lies between 1 and 31.

**Imports**

*import java.utils.Date;*

**Returns**

The day of the month with a value between 1 and 31. Return type is int.

**See Also**

The setDate method in the Date class

**Example**

Refer to Listing 13-4. In the today() method getDate is used to find today's date.

## getDay()

**ClassName**

Date

**Purpose**

Returns the day of the week in a Date object.

**Syntax**

public int getDay()

**Parameters**

None.

**Description**

This method returns the day of the week. The returned value lies between 0-6. Sunday is 0 and Saturday is day 6.

**Imports**

*import java.utils.Date;*

**Returns**

The day of the week with a value between 0 and 6. Return type is int.

**See Also**

The setDate method in the Date class

**Example**

Refer to Listing 13-4. This method is used in the today() method to find out what day today is. This is done with assistance from the whatDay method.

## getHours()

**ClassName**

Date

**Purpose**

Returns the hour of the day in a Date object.

**Syntax**

public int getHours()

**Parameters**

None.

**Description**

This method returns the hour of the day. The returned value lies between 0 and 23. Midnight is 0 hours, noon is 12, and 23 indicates 11pm.

**Imports**

*import java.utils.Date;*

**Returns**

The hour of the day with value between 0 and 23. Return type is int.

**See Also**

The setHours method in theDate class

**Example**

Refer to Listing 13-4. The now()method in the demoDate class uses the getHours method to find the hours passed for the current day.

## getMinutes()

**ClassName**

Date

**Purpose**

Returns the minutes in a Date object.

**Syntax**

public int getMinutes()

**Parameters**

None.

**Description**

This method returns the minutes past an hour. The returned value lies between 0 and 59.

**Imports**

*import java.utils.Date;*

**Returns**

The minutes past an hour with a value between 0 and 59. Return type is int.

**See Also**

The setMinutes method in the Date class

**Example**

Refer to Listing 13-4. In the now() method, the getMinutes method is used to find the minutes past the hour.

## getMonth()

**ClassName**

Date

**Purpose**

Returns the month of the year in a Date object.

**Syntax**

public int getMonth()

**Parameters**

None.

**Description**

This method returns the month of the year. The returned value lies between 0 and 11. January is month 0, February is month 1, December is month 11, and so on.

**Imports**

*import java.utils.Date;*

**Returns**

The month of the year. Return type is int.

**See Also**

The setMonth method in the Date class

**Example**

Refer to Listing 13-4. In the today() method, the getMonth method is used to find the month of the year and mapping to strings is accomplished by using the whatDay method.

## getSeconds()

**ClassName**

Date

**Purpose**

Returns the seconds past a minute in a Date object.

**Syntax**

public int getSeconds()

**Parameters**

None.

**Description**

This method returns the seconds past a minute. The returned value lies between 0 and 59.

**Imports**

*import java.utils.Date;*

**Returns**

The seconds past a minute with a value between 0 and 59. Return type is int.

**See Also**

The setSeconds method in the Date class

**Example**

Refer to Listing 13-4. In the now() method, the getSeconds method is used to find the seconds.

## getTime()

**ClassName**

Date

**Purpose**

Returns the number of milliseconds elapsed since epoch.

**Syntax**

public long getTime()

**Parameters**

None.

**Description**

This method returns the number of milliseconds elapsed since the epoch. This method can be used to find the timings of program execution by finding the start and end of the execution.

**Imports**

*import java.utils.Date;*

**Returns**

The milliseconds elapsed since epoch. Return type is long.

**See Also**

The setTime method in the Date class

**Example**

Refer to Listing 13-4. In the timeTheMethod method, the time taken to execute 10,000 null loops is found using this method.

## getTimezoneOffset()

**ClassName**

Date

**Purpose**

Returns the time zone offset between the local time and the standard time, denoted in minutes.

**Syntax**

public int getTimezoneOffset()

**Parameters**

None.

**Description**

This method returns the difference in time between the local time and the standard Universal Time. The offset is calculated and returned in minutes.

**Imports**

*import java.utils.Date;*

**Returns**

The difference in time zones in units of minutes. Return type is int.

**Example**

Refer to Listing 13-4. In the now() method, the getTimezoneOffset method is used to find the difference in time, between the local time and the standard time, in units of minutes.

## getYear()

**ClassName**
Date
**Purpose**
Returns the number of years from 1900 till the current day.
**Syntax**
public int getYear()
**Parameters**
None.
**Description**
This method returns the number of years from 1900 till the current day. For example, if the current year is 1997, then this method will return a value of 97.
**Imports**
*import java.utils.Date;*
**Returns**
The years since 1900. Return type is int.
**See Also**
The setYear method in the Date class
**Example**
Refer to Listing 13-4. In the now() method, the getYear method is used to find the number of years from 1900 till the current day.

## hashCode()

**ClassName**
Date
**Purpose**
Obtains the hash code of the target Date object.
**Syntax**
public int hashCode()
**Parameters**
None.
**Description**
The hash code of the target date object is returned by this method. It overrides the hashCode method of the Object class.
**Imports**
*import java.utils.Date;*
**Returns**
The hash code of the target object. Return type is int.
**Example**
Refer to the now()method in Listing 13-4. The hashCode method is used to debug the object information.

## parse(String)

**ClassName**
Date

**Purpose**

Parses the given time in string format and converts it to time value.

**Syntax**

public static long parse(String *date_string*)

**Parameters**

***date_string***

The date in one of the string formats.

**Description**

This method parses the date that is specified in String format, using one of the standard date specifying conventions. It returns the number of milliseconds elapsed since epoch. This is a static method and, hence, can be invoked directly by using the class name without creating a date object for this purpose

**Imports**

*import java.utils.Date;*

**Returns**

The milliseconds elapsed since epoch for the specified string representation of date. Return type is long.

**See Also**

The toString, toGMTString, and toLocalString methods in the Date class

**Example**

Refer to Listing 13-4. The getTimeFromString method takes the string as input and returns the time. It uses this parse method to accomplish the same.

## setDate(int)

**ClassName**

Date

**Purpose**

Sets the day of the month in a Date object.

**Syntax**

public void setDate(int day)

**Parameters**

***day***

The day of the month, with values between 1 and 31.

**Description**

This method sets the day of the month. The value specified lies between 1 and 31.

**Imports**

*import java.utils.Date;*

**Returns**

None.

**See Also**

The getDate method in the Date class

**Example**

Refer to Listing 13-4. In the demoUTCtimes method, setDate is used to fix the date of the Date object.

## setHours(int)

**ClassName**
Date
**Purpose**
Sets the hour of the day in a Date object.
**Syntax**
public void setHours(int hours)
**Parameters**
***hours***
The hour that the date object is to be set to.
**Description**
This method sets the hour of the day. The specified value lies between 0 and 23. Midnight is 0 hours, noon is 12, and 23 indicates 11pm.
**Imports**
*import java.utils.Date;*
**Returns**
None.
**See Also**
The getHours method in the Date class
**Example**
Refer to Listing 13-4. The demoUTCtimes()method, in the demoDate class, uses this setHours method to fix the hours passed on the day.

## setMinutes(int)

**ClassName**
Date
**Purpose**
Sets the minutes in a Date object.
**Syntax**
public void setMinutes(int mins)
**Parameters**
***mins***
The number of minutes past an hour that is to be set as the minutes in the Date object.
**Description**
This method sets the minutes past an hour for the target Date object. The value lies between 0 and 59.
**Imports**
*import java.utils.Date;*
**Returns**
None.
**See Also**

The getMinutes method in the Date class

**Example**

Refer to Listing 13-4. In the demoUTCtimes() method, the setMinutes method is used to fix the minutes past the hour.

## setMonth(int)

**ClassName**

Date

**Purpose**

Sets the month of the year in a Date object.

**Syntax**

public void setMonth(int month)

**Parameters**

***month***

Month of the year that is to be set as the month in the target Date object.

**Description**

This method sets the month of the year in the target Date object. The value lies between 0 and 11. January is month 0, February is month 1, December is month 11, and so on.

**Imports**

*import java.utils.Date;*

**Returns**

None.

**See Also**

The getMonth method in the Date class

**Example**

Refer to Listing 13-4. In the demoUTCtimes()method, the setMonth method is used to fix the month of the year of the target Date object.

## setSeconds(int)

**ClassName**

Date

**Purpose**

Sets the seconds past a minute in a Date object.

**Syntax**

public void setSeconds(int secs)

**Parameters**

***secs***

The number of seconds elapsed since the start of the present minute.

**Description**

This method sets the seconds past the last minute in the target Date object. The value used lies between 0 and 59.

**Imports**

*import java.utils.Date;*

**Returns**

None.

**See Also**

The getSeconds method in the Date class

**Example**

Refer to Listing 13-4. In the demoUTCtimes() method, the setSeconds method is used to fix the number of seconds in the Date object.

## setTime(long)

**ClassName**

Date

**Purpose**

Sets the number of milliseconds elapsed since epoch.

**Syntax**

public void setTime(long time)

**Parameters**

***time***

The number of milliseconds since epoch.

**Description**

This method sets the number of milliseconds elapsed since the epoch. This effectively sets the date (year, month, day, hours, minutes, seconds).

**Imports**

*import java.utils.Date;*

**Returns**

None.

**See Also**

The getTime method in the Date class

**Example**

Refer to Listing 13-4. In the demoUTCtimes method, this method is used to set the time of the date object whose UTC time is already found by using the UTC method in class Date.

## setYear(int)

**ClassName**

Date

**Purpose**

Sets the number of years from 1900 till the specified day.

**Syntax**

public void setYear(int year)

**Parameters**

***year***

The number of years from 1900 for the target Date object.

**Description**

This method fixes the number of years from 1900 to be the year attribute of the target Date object. For example, if you want to set the new date to be in 1946, specify 46 as the parameter to this method. If the year specified is out of bounds (if you specify 1997 instead of 97), then an IllegalArgumentException is thrown.

**Imports**

*import java.utils.Date;*

**Returns**

None.

**See Also**

The getYear method in the Date class

**Example**

Refer to Listing 13-4. This setYear method is used in method demoUTCtimes to fix the year.

## toGMTString()

**ClassName**

Date

**Purpose**

Using the GMT conventions, this method converts the date to a string.

**Syntax**

public String toGMTString()

**Parameters**

None.

**Description**

This method converts the target date object into a string representation using the GMT convention. This GMT convention string is returned by this method.

**Imports**

*import java.utils.Date;*

**Returns**

The GMT convention string representation of the date. Return type is String.

**See Also**

The toString and toLocaleString methods in the Date class

**Example**

Refer to the now()method in Listing 13-4. The two conventions of time are printed to the screen using this toGMTString method with another method, toLocaleString.

## toLocaleString()

**ClassName**

Date

**Purpose**

Using the Locale conventions, this method converts the date to a string.

**Syntax**

public String toLocaleString()

**Parameters**

None.

**Description**

This method converts the target date object into a string representation using the locale convention. This locale convention string is returned by this method.

**Imports**

*import java.utils.Date;*

**Returns**

The locale convention string representation of the date. Return type is String.

**See Also**

The toString and toGMTString methods in the Date class

**Example**

Refer to now()method in Listing 13-4. The two conventions of time are printed to screen using this toLocaleString method with another method, toGMTString.

## toString()

**ClassName**

Date

**Purpose**

Using the standard unix ctime conventions, this method converts the date to a string.

**Syntax**

public String toString()

**Parameters**

None.

**Description**

This method converts the target date object into a string representation, using the standard Unix ctime string format convention. This ctime convention string is then returned by this method.

**Imports**

*import java.utils.Date;*

**Returns**

The Unix ctime convention string representation of the date. Return type is String.

**See Also**

The toGMTString and toLocaleString method in the Date class

**Example**

Refer to the main()method in Listing 13-4. The details of the date are printed to the standard output terminal using this method.

## Display Timer Application

The project for this chapter is a display timer application. It consists of a central timer, which keeps track of the time, and two clock objects that obtain the time information

from the central timer and display it in their own style. You will use the Date object to obtain time information. This application will give you some practice in designing entities that are dependent on another object for information. The main goal of developing this application is to become familiar with the Observable-Observer design pattern, which results in low coupling between objects.

First you develop a clockTimer class that will serve as the central timer. This class must subclass the Observable class, because it will be monitored by other clock classes. This observable object will notify other observer objects of each notable change in time. The minimum unit of time you will work with in this application is seconds. When the seconds value changes, the observer objects (clocks, in this case) will be notified of the change.

You'll develop two types of clock: a digital clock and an analog clock, which differ in the way they display the time obtained from the central timer. These clock classes, DigitalClock and AnalogClock, implement the Observer interface, so they can register themselves as observers of the central timer. Implementing the interface means that these clock classes have to define the update method of the Observer interface, so that it updates the display of time in their own style of display. The AnalogClock will display the time with three hands: the hour hand, the minute hand, and the second hand. DigitalClock will display the hour, minutes, and seconds as numerals.

You will develop these two clocks as separate frames that display the time. When the DigitalClock and AnalogClock are constructed, they will register themselves as observers of the clock timer. Whenever there is a notable change in time, the clockTimer will notify all its observers, and these observer objects will then display the time independently in their own style.

In this implementation, you'll develop Threads for each object, so you will have a Thread that keeps track of the time and notifies the observers of the change in time. The observer Threads will keep displaying the current time.

Apart from these classes, you shall have a wrapper class to make these three objects cooperate with each other in displaying the time. This will be displayTimer class, which will create instances of the clock and timer classes and start all the clocks.

## *Building the Project*

**1.** The first step is to implement the clockTimer class, which subclasses the Observable class. This is also a Thread that keeps track of the time, so this class implements the Runnable interface. This class will have hours, minutes, and seconds as data members. On construction, it will find the current time and assign the values to its members. It will have accessor methods to return the values of these members, when they are requested by other objects. The following code implements these features.

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

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

class ClockTimer extends Observable implements Runnable {

        int hour;
        int mins;
        int secs;
     //   Date date;

        public ClockTimer() {
                Date date = new Date();
                hour = date.getHours();
                mins = date.getMinutes();
                secs = date.getSeconds();
                start();
        }

        public int GetHours() {
                return hour;
        }

        public int GetMinutes() {
                return mins;
        }

        public int GetSeconds() {
                return secs;
        }
}
```

**2.** The clockTimer class implements the Runnable interface and also calls the start() method to execute the next step. This method instantiates a Thread for this class. It can pass itself as the target for creating the Thread, because clockTimer implements the Runnable interface. You must also define the run() method because it is the method in the Runnable interface. The run() method will be called by the Thread that is instantiated in the start() method. Enter the following code in the clockTimer class.

```
Thread timeThread; // thread member of the class

public void start() {
       if (timeThread == null) {
               timeThread = new Thread(this);
               timeThread.start();
       }
}

public void run() {
        while (timeThread != null) {
                try {
                        tick();
                       Thread.sleep(1000);

                } catch (InterruptedException e){
                             debug("exception");
```

```
                                }
        }
}
```

**3.** In the run() method, for every 1 second (1000 milliseconds of sleep) a call is made to a method name tick(). This method updates the clockTimer on every tick (1 second) of the new time (hours, minutes, and seconds). It notifies all the observers in the list of observers of this observable object by calling the notifyObservers() method of the Observable class. Enter the following code inside the clockTimer class to complete the implementation.

```
public synchronized void tick() {

          clearChanged();  // clear the flag
          Date date = new Date(); // find the current time
          if (date.getSeconds() != secs) {
              // if there is a change
                  setChanged();    // set the flag
                 // update the values of the members
                  hour = date.getHours();
                  mins = date.getMinutes();
                  secs = date.getSeconds();
          }
          notifyObservers(); // notify all the observers
of the change
}
```

**4.** The next class you'll implement is the DigitalClock class. It is an Observer class that has a Thread for updating the display. Also, this will form a separate frame to display the time. Hence, the DigitalClock class will subclass the Frame class and implement both Observer and Runnable interfaces. It has members to keep track of the time and to display the time on the frame. It also has an Observable object of type clockTimer as a member. This member object is required to access the time after notification. This implementation uses a pull-style design where the Observable notifies the observers and the observers ask for more information from the Observable. Now enter this code for the class DigitalClock.

```
class DigitalClock extends Frame implements Observer, Runnable {

        ClockTimer myTimer;
        int hours;
        int mins;
        int secs;

        public DigitalClock(ClockTimer timer) {
                myTimer = timer;
                hours = myTimer.GetHours();
                mins = myTimer.GetMinutes();
                secs = myTimer.GetSeconds();
                myTimer.addObserver(this);
                setTitle("Digital");
                resize(200,100);
                show();
                start();
        }
}
```

**5.** The Thread creation and the run method are also implemented for the DigitalClock class. These tasks are done in the start() and run() methods of the class. Enter the following code inside the DigitalClock class definition.

```
Thread clockThread;

  public void start() {
          if (clockThread == null) {
                  clockThread = new Thread(this);
                  clockThread.start();
          }
  }
  public void run() {

          while (clockThread != null) {
              repaint();
              try {
                  clockThread.sleep(1000);
              } catch (InterruptedException e){
              }
          }
}
```

**6.** Now you've implemented the methods for object creation and Thread initiation. The DigitalClock implements the Observer interface, so you need to define the update() method of the interface, which is called when the Observable makes a call to the notifyObservers() method. If a call to update is received, it means that a change that interests the class may have occurred, so the Observable member object is queried for the change. After noting the change, the frame is repainted with its new values. This is done by overriding the paint method in the DigitalClock class. After setting a black background, the object displays the time in green numerals. Enter the following two methods, update and paint, inside the DigitalClock class.

```
public void update(Observable obs, Object arg) {
             if (obs.hasChanged()) {
                     myTimer = null;
                     myTimer = (ClockTimer)obs;
                     hours = myTimer.GetHours();
                     mins = myTimer.GetMinutes();
                     secs = myTimer.GetSeconds();
             }
             repaint();
     }

public void paint(Graphics g) {
           String secs_str;
           if (secs < 10)
                   secs_str = "0"+secs;
           else
                   secs_str = ""+secs;
           String s = new String(" "+  hours + ":" + mins+ ":"
           +secs_str);
           Dimension d = size();
           g.setColor(Color.black);
           g.fillRect(0,0, d.width, d.height);
           g.setColor(Color.green);
```

```
            g.setFont(new Font("TimesRoman", Font.BOLD,20));
            g.drawString(s, 50, 70);
      }
```

**7.** The AnalogClock you are going to implement is similar to the DigitalClock, except that the display changes. They differ mainly in the way they repaint the screen. Note that implementing the two clocks separately may not be the best design. Alternatively, you could implement a generic Clock class and subclass the class as DigitalClock and AnalogClock class. But we are ignoring design criteria here to focus on implementation issues. Enter the following code to implement the AnalogClock class.

```
class AnalogClock extends Frame implements Observer, Runnable {

        ClockTimer myTimer;
        int hours;
        int mins;
        int secs;

            public AnalogClock(ClockTimer timer) {
            myTimer = timer;
            hours = myTimer.GetHours();
            mins = myTimer.GetMinutes();
            secs = myTimer.GetSeconds();
            myTimer.addObserver(this);
            resize(150,150);
            setTitle("Analog");
            start();
            show();
      }

      Thread clockThread;

      public void start() {
              if (clockThread == null) {
                      clockThread = new Thread(this);
                      clockThread.start();
              }
      }

      public void run() {

            while (clockThread != null) {
                    repaint();
                    try {
                            clockThread.sleep(1000);
                    } catch (InterruptedException e){}
            }
      }
}
```

**8.** The update method of the Observer interface has to be defined in the AnalogClock class. It updates the members of this class and makes a call to repaint the frame so it displays the new time in analog style.

```
public void update(Observable obs, Object arg) {
        if (obs.hasChanged()) {
                myTimer = null;
                myTimer = (ClockTimer)obs;
```

```
                hours = myTimer.GetHours();
                mins = myTimer.GetMinutes();
                secs = myTimer.GetSeconds();
        }
        repaint();
}
```

**9.** The purpose of calling to repaint in the update(Observable, Object) method is to repaint the frame. We want to display the time using an hour hand, a minutes hand, and a second hand. The hands change their position depending on the values of the member values. We keep track of the various positions of x and y coordinates for these hands, first drawing the earlier visible hands with the background color and then drawing the hands corresponding to the new values of hours, minutes, and seconds with the red color. The present x and y positions, as well as the previous positions, are maintained as data members of this class. Enter the following code inside the AnalogClock class definition.

```
int xpos;
int ypos;
int lastxpos;
int lastypos;

int Xhours;
int Xmins ;
int Xsecs;

  public void update(Graphics g) {

double x = xpos - 75;
double y = ypos - 75;
double lastx = lastxpos - 75;
double lasty = lastypos - 75;
double angle = Math.atan(-x/y);
double lastangle = Math.atan(-lastx/lasty);
Dimension d = size();
g.setColor(Color.white);
g.fillRect(0,0,d.width,d.height);
g.drawLine(75, 75,
 (int)(75+50*Math.sin((double)(2*3.145*(Xsecs)/60))),
 (int)(75-0*Math.cos((double)(2*3.145*(Xsecs)/60))));
if (secs == 0) {
  g.drawLine(75, 75,
   (int)(75+50*Math.sin((double)(2*3.145*Xmins/60))),
   (int)(75-0*Math.cos((double)(2*3.145*Xmins/60))));
  g.drawLine(75, 75,
   (int)(75+40*Math.sin((double)(2*3.145*(Xhours+Xmins/60.0)
/12))),
   (int)(75-(0*Math.cos((double)(2*3.145*(Xhours+Xmins/60.0)
/12))));
  }
  g.setColor(Color.red);
Xhours = hours;
Xmins = mins;
Xsecs = secs;
g.drawLine(75, 75,
      (int)(75+50*Math.sin((double)(2*3.145*secs/60))),
      (int)(75-50*Math.cos((double)(2*3.145*secs/60))));
```

```
g.setColor(Color.black);
g.drawLine(75, 75,
      (int)(75+50*Math.sin((double)(2*3.145*mins/60))),
      (int)(75-50*Math.cos((double)(2*3.145*mins/60))));
g.drawLine(75, 75,
          (int)(75+40*Math.sin((double)(2*3.145*(hours+mins/60.0)
 /12))),
          (int)(75-
40*Math.cos((double)(2*3.145*(hours+mins/60.0)/12)
 )));
 if (ypos > 75)
          angle += 3.145;
 if (lastypos > 75)
          lastangle += 3.145;
}
```

**10.** Now that you have implemented the three classes for your application, all that remains is to implement the displayTimer class, which will act as a driver to run the three threads that display the time in digital and analog styles. This will be the public class in the implementation. Because the displayTimer will run as a stand-alone application, implementation of the main method is essential. Enter all the classes you created earlier and the following class in a file named displayTimer.java.

```
public class displayTimer extends java.applet.Applet    {
        ClockTimer ct;
        DigitalClock dc;
        AnalogClock ac;

        public void init() {
                ct = new ClockTimer();
                dc = new DigitalClock(ct);
                ac = new AnalogClock(ct);
        }

        public void start() {
                ct.run();
        }

        public final static void main(String args[]) {
                displayTimer dt = new displayTimer();
                dt.init();
                dt.start();
        }
}
```

When the displayTimer.java file is compiled and the class file is executed using the Java interpreter, the resultant frames display the analog and the digital clocks, as shown in Figure 13-9.

**Figure 13-9** Digital and analog clocks in action

### *How It Works*

When you start the application using the Java interpreter by typing java displayTimer, two frames appear on the screen. The frame titled Digital displays the current time in digital style, where the hours, minutes, and seconds are separated by colons. In Figure 13-9, the time displayed in the Digital frame is 23:11:16, which is 11 minutes and 16 seconds past 11 pm. The frame titled Analog displays the same time as shown in the digital clock, in analog style using three hands: hours, minutes, and seconds. Both of these clocks are Observer objects observing the time from a central time server object, which acts as an Observable. It's time to end this chapter!